Laurence Labayen
Nov 18, 2019

Lab #6

CS2302 Data Structures - MW 10:30am

Professor: Dr. Olac Fuentes
T.A.: Anindita Nath

Fall 2019

Description:

This lab was about implementing graphs and using them to solve the problem of the fox, chicken, grain and farmer. We were asked to create functions to insert edge, delete edge and display for adjacency matrix and edge list implementations. Additionally, we were tasked with functions to convert graph implementations to two other graphs (as_AL, as_AM, as_EL). For the second part, we had to solve a problem that consists of a fox, chicken, grain and farmer. The farmer has to cross all three to the other side of the river one at a time and bound by the following rules. The fox cannot be left with the chicken and the chicken cannot be left with the grain because it will it eat it.

Example from instructions for part 2:

$(\langle 0, 0, 0, 0\rangle, \langle 0, 1, 0, 1\rangle) \in E$, *corresponding to the person crossing the river with the chicken*

$(\langle 0, 0, 0, 0\rangle, \langle 0, 0, 1, 1\rangle) \notin E$, *since* $\langle 0, 0, 1, 1\rangle$ *is not a legal state (the fox eats the chicken)*

Solution design and Implementation:

Our first task was to implement insert edge on two of the other graphs (adjacency matrix and edge list). Given the source, destination and weight of inserting an edge. We must first check if the parameters are out of range. Another check if the weight being passed is not 1 if the graph is unweighted. If none of the checks are validated, we add the edge to our graph. For undirected graphs, we must also add the inverse of the source and destination as well. Next is deleting an edge. Given the source and destination with an adjacency matrix graph, we can use these as indices to find the position of the edge being removed or set to the default weight (-1). Again, with an undirected graph, we must also delete the inverse of the given source and weight. With an edge list graph, we must traverse the list and find the source and destination of the edge being deleted. To display the adjacency matrix graph, we simply print the entire list of edges. For the edge list, we have to traverse all the edges and print them.

To tackle the conversion of graph representation, we must first create a graph of the desired representation and use the length of the current graph as the length of the new graph. After, we just use a loop to traverse the current graph and insert into the new graph. A nested for loop must be used for the converting the adjacency list and adjacency matrix to other graphs. Next is the draw function, as per the instructions, we were allowed to use the as_AL function to convert any graph to an adjacency list and be used on the provided draw function.

Part two of the lab consists of the solving the problem of the fox, chicken and grain using breadth first search and depth first search. After solving the problem on paper, I came up with 2 solutions using 4-bits ⟨b0,b1,b2,b3⟩ and 8 steps to cross the fox, chicken and grain across or going from ⟨0,0,0,0⟩ to ⟨1,1,1,1⟩ with the constraints given to us. Thereafter, I used a total of 8 edges of the solution as input for my graph program with a size of 16. Next step was to implement the breadth first search function to solve to problem. Passing the parameters start and end, the first step is to create a visited list and set all to False. Next, we need to make a queue and append the start vertex into it. We go into a loop as long as there is a queue, in the loop, we pop an element and assign it to start. Then, we get all adjacent vertices of the popped vertex s. If the adjacent vertex has not been visited, then mark it visited and append it to the queue. If the end vertex is found, we return. For the depth first search, we use recursion and a helper function. First, we pass the start and end vertices to the main function and the main function will create a list of visited vertices and call the helper function. In this function, we check if start is in the visited list, then check if visited is not empty and if the last element of visited is the end vertex. Then, call itself recursively with the starting element as the destination of the neighbors of start. Additionally, I added two extra functions, path_steps function that will convert the path from the breadth first search or depth first search to binary format ⟨b0,b1,b2,b3⟩ as per lab instructions and draw_path function to draw a highlighted path from start to end vertices.

```python
# Insert edge function that adds an edge given its source, destination and weight
def insert_edge(self,source,dest,weight=1):

    # check parameters are valid
    if source >= len(self.am) or dest>=len(self.am) or source <0 or dest<0:
        print('Error, vertex number out of range')
    elif weight!=1 and not self.weighted:
        print('Error, inserting weighted edge to unweighted graph')
    else:
        # check if not directed
        if not self.directed:
            # insert weight to both valid positons for undirected
            self.am[source][dest]=weight
            self.am[dest][source]=weight

            # otherwise, graph is directed and must only be inserted one way
        else:
            self.am[source][dest]=weight
    return

# Delete edge function that deletes an edge given its source and destination
def delete_edge(self,source,dest):

    # check if parameters are valid
    if source >= len(self.am) or dest>=len(self.am) or source <0 or dest<0:
        print('Error, vertex number out of range')
    else:
        # check if directed or undirected
        if not self.directed:
            # if undirected, delete or subsitute to -1 both positions
            self.am[source][dest]=-1
            self.am[dest][source]=-1
            # otherwise for directed, only one position is deleted
        else:
            self.am[source][dest]=-1
    return

# Display function to print the entire adjacency matrix edges
def display(self):
    print(self.am)
```

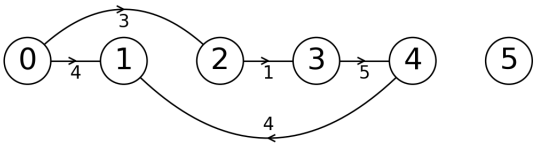
```python
# Insert edge function that adds an edge given its source, destination and weight
def insert_edge(self,source,dest,weight=1):

    if weight!=1 and not self.weighted:
        print('Error, inserting weighted edge to unweighted graph')
    else:
        # insert edge to graph
        self.el.append(Edge(source,dest,weight))

# Delete edge function that deletes an edge given its source and destination
def delete_edge(self,source,dest):

    # find position of the given edge and remove from graph
    for i in self.el:
        if i.source==source and i.dest==dest:
            self.el.remove(i)

# Displays all the edges in the graph
def display(self):
    print('[',end='')
    for i in self.el:
        print('('+str(i.source)+','+str(i.dest)+','+str(i.weight)+')',end='')
    print(']',end=' ')
    print('')

# Draw function that converts the current graph to an adjacency list then draws
# the graph (as per lab instructions)
def draw(self):

    # converts current graph to an adjacency list to be used for the function
    adjlist = self.as_AL()

    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(adjlist.al)):
        for edge in adjlist.al[i]:
            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
```

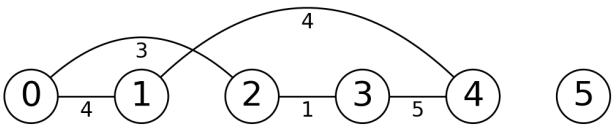
```python
# Breadth first search function used to return path
def BFS(self, s,end):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.al))

    # Create a queue for BFS
    queue = [[s]]

    while queue:

        # pop element from queue and assign it to s
        s = queue.pop(0)

        # if end is found, return
        if s[-1]==end:
            print('From AL BFS')
            return s


        # Get all adjacent vertices of the
        # popped vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in self.al[s[-1]]:
            if visited[i.dest] == False:
                queue.append(s + [i.dest])
                visited[i.dest] = True
```

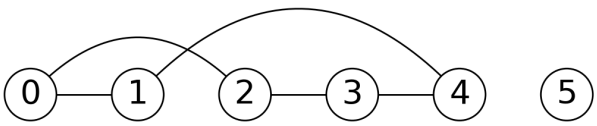
```python
# Depth first search function used to return path
def DFS(self, s, end):
    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From AL DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:
        # check if visited is not empty and if the last element of
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return
        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for neighbour in self.al[s]:
            self.DFS_(visited, neighbour.dest, end)

    return visited
```

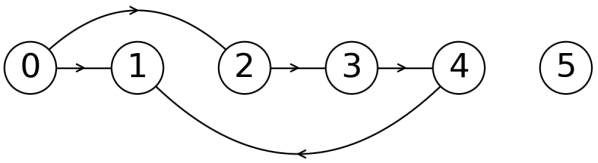
Experimental Results:

test_graph.py:









Adjacency List

```
You selected Adjacency List

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

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

as_AM
[[-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]]

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

Adjacency Matrix:

```
You selected Adjacency Matrix

[[-1  1  1 -1 -1 -1]
 [ 1 -1  1 -1  1 -1]
 [ 1  1 -1  1 -1 -1]
 [-1 -1  1 -1  1 -1]
 [-1  1 -1  1 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
[[-1  1  1 -1 -1 -1]
 [ 1 -1 -1 -1  1 -1]
 [ 1 -1 -1  1 -1 -1]
 [-1 -1  1 -1  1 -1]
 [-1  1 -1  1 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
[[-1  1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1]
 [-1 -1 -1  1 -1 -1]
 [-1 -1 -1 -1  1 -1]
 [-1  1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
[[-1  1  1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1]
 [-1 -1 -1  1 -1 -1]
 [-1 -1 -1 -1  1 -1]
 [-1  1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
```

```
[[-1  4  3 -1 -1 -1]
 [ 4 -1  2 -1  4 -1]
 [ 3  2 -1  1 -1 -1]
 [-1 -1  1 -1  5 -1]
 [-1  4 -1  5 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
[[-1  4  3 -1 -1 -1]
 [ 4 -1 -1 -1  4 -1]
 [ 3 -1 -1  1 -1 -1]
 [-1 -1  1 -1  5 -1]
 [-1  4 -1  5 -1 -1]
 [-1 -1 -1 -1 -1 -1]]
[[-1  4  3 -1 -1 -1]
 [-1 -1  2 -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]]
[[-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]]
```

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

as_AM
[[-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]]

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

Edge List:

```
You selected Edge List

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

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

as_AM
[[-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]]

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

<u>Fox, Chicken and Grain:</u>

Adjacency List:

```
You selected Adjacency List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 1
From AL BFS
[0, 5, 4, 7, 11, 10, 15]

From AL BFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
11 [1, 0, 1, 1]
10 [1, 0, 1, 0]
15 [1, 1, 1, 1]

Search running time: 0.000109 seconds
```

```
You selected Adjacency List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 2
From AL DFS
[0, 5, 4, 7, 11, 10, 15]

From AL DFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
11 [1, 0, 1, 1]
10 [1, 0, 1, 0]
15 [1, 1, 1, 1]

Search running time: 9.1e-05 seconds
```

Adjacency Matrix:

```
You selected Adjacency Matrix

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 1
From AM BFS
[0, 5, 4, 7, 11, 10, 15]

From AM BFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
11 [1, 0, 1, 1]
10 [1, 0, 1, 0]
15 [1, 1, 1, 1]

Search running time: 0.000218 seconds
```

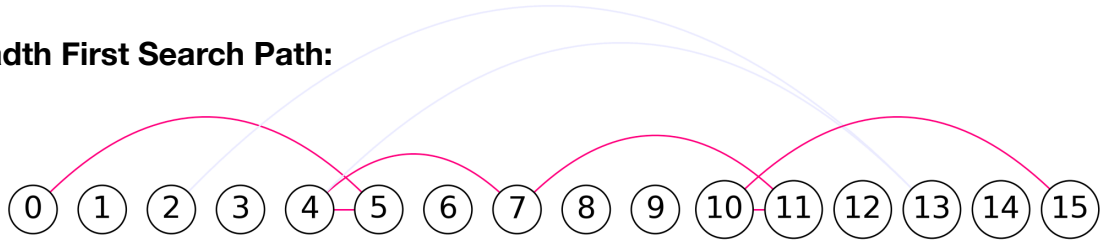
```
You selected Adjacency Matrix

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 2
From AM DFS
[0, 5, 4, 7, 11, 10, 15]

From AM DFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
11 [1, 0, 1, 1]
10 [1, 0, 1, 0]
15 [1, 1, 1, 1]

Search running time: 0.000254 seconds
```

Edge List:

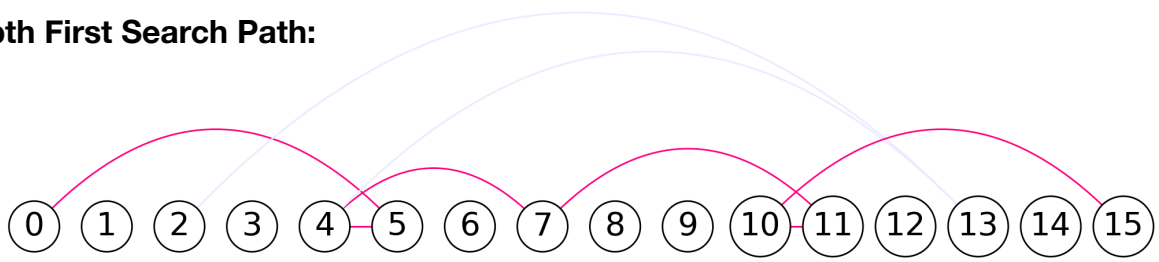
```
You selected Edge List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 1
From EL BFS
[0, 5, 4, 7, 13, 11, 15]

From EL BFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
13 [1, 1, 0, 1]
11 [1, 0, 1, 1]
15 [1, 1, 1, 1]

Search running time: 0.000137 seconds
```

```
You selected Edge List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 2

1. Breadth First Search
2. Depth First Search

Select search function: 2
From EL DFS
[0, 5, 4, 7, 13, 11, 15]

From EL DFS
0 [0, 0, 0, 0]
5 [0, 1, 0, 1]
4 [0, 1, 0, 0]
7 [0, 1, 1, 1]
13 [1, 1, 0, 1]
```

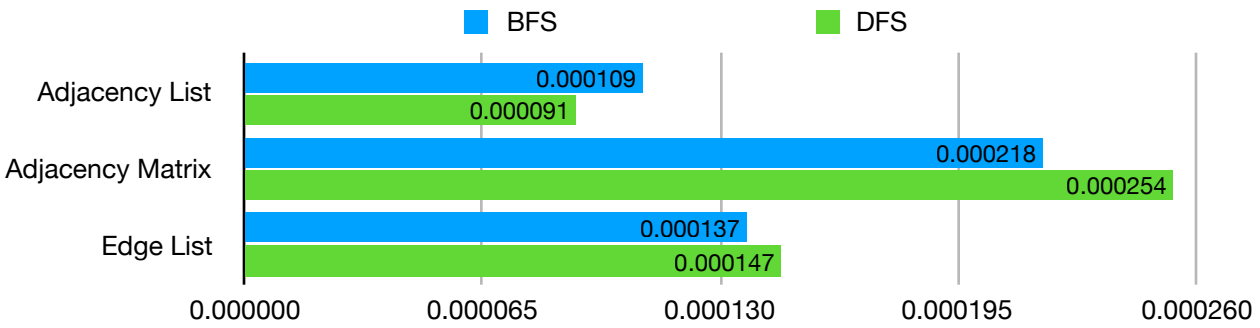
Breadth First Search Path:



Depth First Search Path:



Fox, Chicken and Grain Search Times:



Insertion Tests:

I tested multiple cases of insertion into different graph implementations with various sizes and various edge numbers.

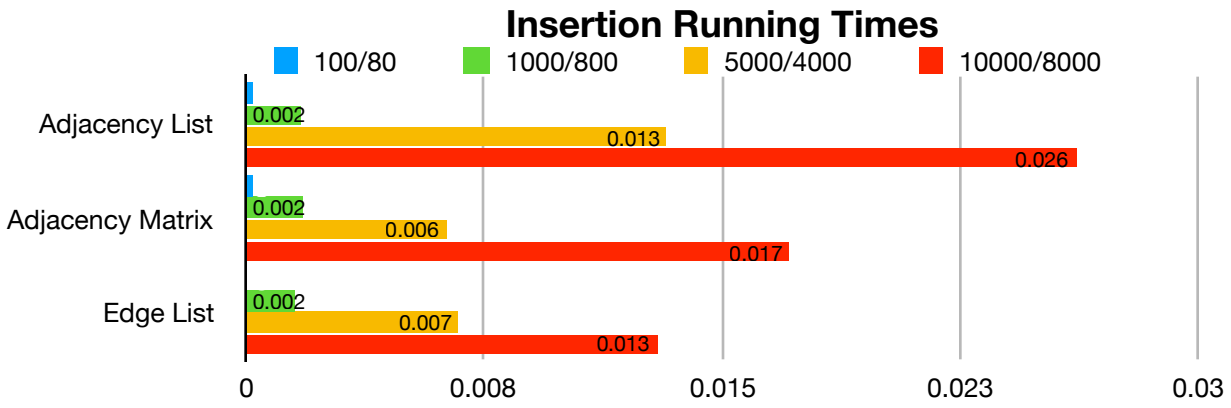
```
You selected Adjacency List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 1000

Enter number of edges to be inserted: 800

Insertion running time with 1000 vertices and 800 edges: 0.001721 seconds
From AL BFS

Breadth First Search running time: 0.0027353 seconds
From AL DFS

Depth First Search running time: 0.0158619 seconds
```

```
You selected Adjacency Matrix

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 1000

Enter number of edges to be inserted: 800

Insertion running time with 1000 vertices and 800 edges: 0.001812 seconds
From AM BFS

Breadth First Search running time: 0.711101 seconds
From AM DFS

Depth First Search running time: 1.3979037 seconds
```

```
You selected Edge List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 1000

Enter number of edges to be inserted: 800

Insertion running time with 1000 vertices and 800 edges: 0.001518 seconds
From EL BFS

Breadth First Search running time: 0.000666 seconds
From EL DFS

Depth First Search running time: 5.327869 seconds
```

Insertion Running Times:

# of vertices/# of edges	Adjacency List	Adjacency Matrix	Edge List
100/80	0.000236	0.000176	7.5E-05
1000/800	0.001721	0.001812	0.001518
5000/4000	0.013263	0.006323	0.006699
10000/8000	0.026144	0.017114	0.012945



Breadth First Search:

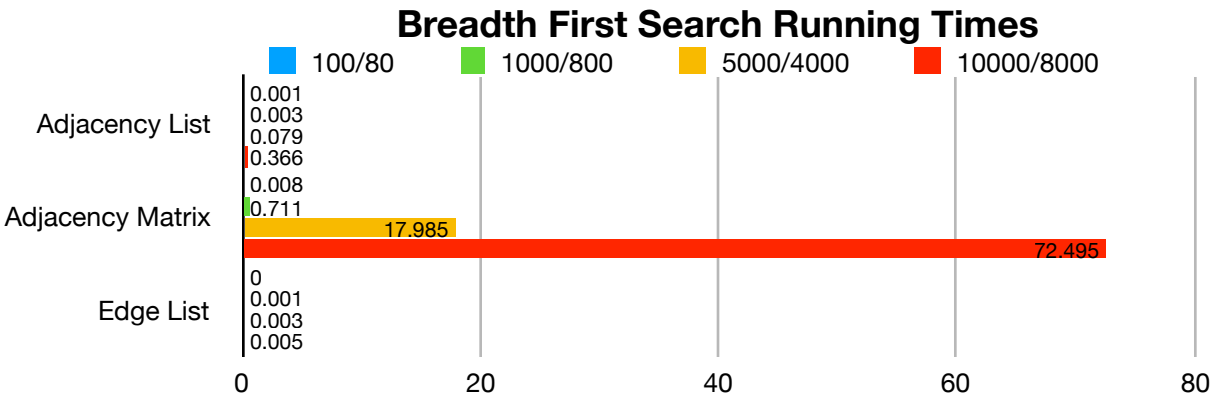
```
You selected Adjacency List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 100

Enter number of edges to be inserted: 80

Insertion running time with 100 vertices and 80 edges: 0.000236 seconds

Would you like to display the edges? (Y/N) n

Would you like to search? (Y/N) y
From AL BFS

Breadth First Search running time: 0.001013 seconds
From AL DFS

Depth First Search running time: 0.001296 seconds
```

```
You selected Adjacency Matrix

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 100

Enter number of edges to be inserted: 80

Insertion running time with 100 vertices and 80 edges: 0.000176 seconds

Would you like to display the edges? (Y/N) n

Would you like to search? (Y/N) y
From AM BFS

Breadth First Search running time: 0.007523 seconds
From AM DFS

Depth First Search running time: 0.01445 seconds
```

```
You selected Edge List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 100

Enter number of edges to be inserted: 80

Insertion running time with 100 vertices and 80 edges: 7.5e-05 seconds
From EL BFS

Breadth First Search running time: 7.61e-05 seconds
From EL DFS

Depth First Search running time: 0.0072947 seconds
```

Breadth First Search Running Times:

# of vertices/# of edges	Adjacency List	Adjacency Matrix	Edge List
100/80	0.001013	0.007523	7.61E-05
1000/800	0.0027353	0.711101	0.000666
5000/4000	0.079044	17.9845367	0.0033451
10000/8000	0.3664234	72.495	0.00535



Depth First Search:

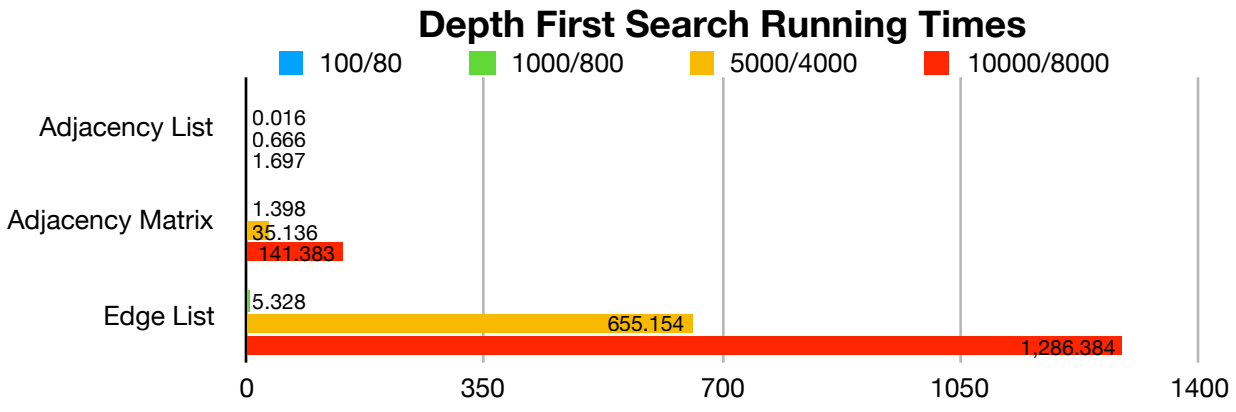
```
You selected Adjacency List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select implementation: 1

Select choice: 3

Enter number of vertices: 5000

Insertion running time with 5000 vertices and 4000 edges: 0.013263 seconds
From AL BFS

Breadth First Search running time: 0.079044 seconds
From AL DFS

Depth First Search running time: 0.6663018 seconds

Enter number of edges to be inserted: 4000
```

```
You selected Adjacency Matrix

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 5000

Enter number of edges to be inserted: 4000

Insertion running time with 5000 vertices and 4000 edges: 0.006323 seconds
From AM BFS

Breadth First Search running time: 17.9845367 seconds
From AM DFS

Depth First Search running time: 35.1355714 seconds
```

```
You selected Edge List

1. Run test_graphs.py
2. Fox, Chicken, Grain, Farmer
3. User selected insertion test

Select choice: 3

Enter number of vertices: 10000

Enter number of edges to be inserted: 8000

Insertion running time with 10000 vertices and 8000 edges: 0.012945 seconds
From EL BFS

Breadth First Search running time: 0.00535 seconds
From EL DFS
```

Depth First Search Running Times:

# of vertices/# of edges	Adjacency List	Adjacency Matrix	Edge List
100/80	0.001296	0.01445	0.0072947
1000/800	0.0158619	1.3979037	5.327869
5000/4000	0.6663018	35.1355714	655.15439
10000/8000	1.6971623	141.383	1286.384



Conclusion:

To conclude this lab, I learned different ways to implement graphs and using those graphs to solve a simple problem. After much testing, I found that some types of graphs don't do so well with search running times, specifically depth first searches. Additionally, I had to increase the recursion depth to a higher limit with testing more than 5000 vertices. Maybe it was my code but the search times were extremely slow for that implementation and parameters. This was a lengthy lab for myself since I didn't really know how to approach the second part.

Appendix:

Main(LAB6.py)

```
import graph_AL as AL
import graph_AM as AM
import graph_EL as EL
import test_graphs as test
import time


print('1. Adjacency List')
print('2. Adjacency Matrix')
print('3. Edge List')


choice=int(input('Select implementation: '))


if choice==1:
    print('\nYou selected Adjacency List')
if choice==2:
    print('\nYou selected Adjacency Matrix')
if choice==3:
```

```
    print('\nYou selected Edge List')



print('\n1. Run test_graphs.py')
print('2. Fox, Chicken, Grain, Farmer')
print('3. User selected insertion test')

choice2=int(input('Select choice: '))

if choice2==1:

    test.tests(choice)

if choice2==2:

    if choice==1:
        g=AL.Graph(16)
    if choice==2:
        g=AM.Graph(16)
    if choice==3:
        g=EL.Graph(16)

    g.insert_edge(0,5)
    g.insert_edge(5,4)
    g.insert_edge(4,7)
    g.insert_edge(4,13)
    g.insert_edge(2,13)
    g.insert_edge(7,11)
```

```
g.insert_edge(10,11)

g.insert_edge(10,15)


timer=0


print('\n1. Breadth First Search')

print('2. Depth First Search')


choice3=int(input('Select search function: '))


if choice3==1:

    start = time.perf_counter()

    print(g.BFS(0,15))

    end = time.perf_counter()

    print('')

    g.path_steps('BFS')


if choice3==2:

    start = time.perf_counter()

    print(g.DFS(0,15))

    end = time.perf_counter()

    print('')

    g.path_steps('DFS')


timer += end - start

print('\nSearch running time:', str(round(timer,6)), 'seconds')

print('\n1. Draw highlighted path')

print('2. Draw complete graph')
```

```
    choice4=int(input('Select Draw function: '))

    if choice4==1:
        #defaults to BFS
        g.draw_path('BFS')
    if choice4==2:
        g.draw()

if choice2==3:
    vert=int(input('Enter number of vertices: '))
    edges=int(input('Enter number of edges to be inserted: '))
    if choice==1:
        g=AL.Graph(vert)
    if choice==2:
        g=AM.Graph(vert)
    if choice==3:
        g=EL.Graph(vert)

    timer=0
    start = time.perf_counter()
    for i in range(edges):
        g.insert_edge(i,i+1)
    end = time.perf_counter()

    timer += end - start
    print('\nInsertion running time with', vert, 'vertices and', edges, 'edges:'
          , str(round(timer,6)),'seconds')
```

```
#    disp=input('Would you like to display the edges? (Y/N) ')
#
#    if disp.lower()=='y' or disp.lower()=='yes':
#        g.display()


#    search=input('Would you like to search? (Y/N) ')


    timer=0


#    if search.lower()=='y' or search.lower()=='yes':
#        search_func=int(input('1 for BFS or 2 for DFS: '))


#    if search_func == 1:
    start = time.perf_counter()
    bfs=g.BFS(0,edges-1)
    end = time.perf_counter()
    timer += end - start
    print('\nBreadth First Search running time:', str(round(timer,7)),'seconds')


#    if search_func == 2:
    start = time.perf_counter()
    bfs=g.DFS(0,edges-1)
    end = time.perf_counter()
    timer += end - start
    print('\nDepth First Search running time:', str(round(timer,7)),'seconds')
```

graph_AL.py

```
# Adjacency list representation of graphs

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import interp1d

import graph_AM as AM

import graph_EL as EL

import sys


sys.setrecursionlimit(150000)


class Edge:

    def __init__(self, dest, weight=1):

        self.dest = dest

        self.weight = weight


class Graph:

    # Constructor

    def __init__(self, vertices, weighted=False, directed = False):

        self.al = [[] for i in range(vertices)]

        self.weighted = weighted

        self.directed = directed


    # Insert Edge function from class website

    def insert_edge(self,source,dest,weight=1):

        if source >= len(self.al) or dest>=len(self.al) or source <0 or dest<0:

            print('Error, vertex number out of range')

        if weight!=1 and not self.weighted:
```

```
        print('Error, inserting weighted edge to unweighted graph')
    else:
        self.al[source].append(Edge(dest,weight))
        if not self.directed:
            self.al[dest].append(Edge(source,weight))
# Delete Edge helper function from class website
def delete_edge_(self,source,dest):
    i = 0
    for edge in self.al[source]:
        if edge.dest == dest:
            self.al[source].pop(i)
            return True
        i+=1
    return False
# Insert Edge function from class website
def delete_edge(self,source,dest):
    if source >= len(self.al) or dest>=len(self.al) or source <0 or dest<0:
        print('Error, vertex number out of range')
    else:
        deleted = self.delete_edge_(source,dest)
        if not self.directed:
            deleted = self.delete_edge_(dest,source)
    if not deleted:
        print('Error, edge to delete not found')
# Display function from class website
def display(self):
    print('[',end='')
    for i in range(len(self.al)):
```

```
        print('[',end='')
        for edge in self.al[i]:
            print('('+str(edge.dest)+','+str(edge.weight)+')',end='')
        print(']',end=' ')
    print(']')
# Draw function from class website
def draw(self):

    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(self.al)):
        for edge in self.al[i]:
            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color='k')
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
```

```
                yd = [y*s for y in yd]
                ax.plot(xd,yd,linewidth=1,color='k')
            if self.weighted:
                xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                yd = [y0[2]-1,y0[2],y0[2]+1]
                yd = [y*s for y in yd]
                ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
        ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)

# as_EL converts current adjacency list graph to an edge list
def as_EL(self):

    # create an empty graph with the same length as current graph
    edgelist = EL.Graph(len(self.al),self.weighted, self.directed)

    # insert edges using a nested loop
    for i in range(len(self.al)):
        for j in self.al[i]:
            edgelist.insert_edge(i, j.dest, j.weight)
    return edgelist
# as_AM converts current adjacency list graph to an adjacency matrix
def as_AM(self):

    # create an empty graph with the same length as current graph
```

```
    matrix =  AM.Graph(len(self.al), self.weighted, self.directed)

    # insert edges using a nested loop
    for i in range(len(self.al)):
        for j in self.al[i]:
            matrix.insert_edge(i, j.dest, j.weight)

    return matrix

def as_AL(self):
    return self

# Breadth first search function used to return path
def BFS(self, s,end):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.al))

    # Create a queue for BFS
    queue = [[s]]

    while queue:

        # pop element from queue and assign it to s
        s = queue.pop(0)

        # if end is found, return
        if s[-1]==end:
```

```
            print('From AL BFS')
            return s


        # Get all adjacent vertices of the
        # popped vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in self.al[s[-1]]:
            if visited[i.dest] == False:
                queue.append(s + [i.dest])
                visited[i.dest] = True

# Depth first search function used to return path
def DFS(self, s, end):
    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From AL DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:
        # check if visited is not empty and if the last element of
```

```
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return
        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for neighbour in self.al[s]:
            self.DFS_(visited, neighbour.dest, end)

    return visited

# Function to print the path in the correct format as shown in the lab
# instructions [b0,b1,b2,b3]
def path_steps(self, func):
    if func == 'DFS':
        search_path = self.DFS(0,len(self.al)-1)
    if func == 'BFS':
        search_path = self.BFS(0,len(self.al)-1)

    for i in search_path:
        print (i, [int(x) for x in list('{0:04b}'.format(i))])

# Modified draw function from class website used to highlight path
#found from BFS or DFS
def draw_path(self, func):
    scale = 30
```

```
fig, ax = plt.subplots()


if func == 'DFS':
    search_path = self.DFS(0,len(self.al)-1)
if func == 'BFS':
    search_path = self.BFS(0,len(self.al)-1)


# create path list to be used to highlight path
path = []
for j in range(len(search_path)-1):
    path.append((search_path[j], search_path[j+1]))




for i in range(len(self.al)):
    for edge in self.al[i]:


        # highlighted path
        if (i, edge.dest) in path or (edge.dest, i) in path:
            line_color = "#ff007f"


        else:
            line_color = "#eeefff"


        d,w = edge.dest, edge.weight
        if self.directed or d>i:
            x = np.linspace(i*scale,d*scale)
            x0 = np.linspace(i*scale,d*scale,num=5)
            diff = np.abs(d-i)
```

```
            if diff == 1:
                y0 = [0,0,0,0,0]
            else:
                y0 = [0,-6*diff,-8*diff,-6*diff,0]
            f = interp1d(x0, y0, kind='cubic')
            y = f(x)
            s = np.sign(i-d)
            ax.plot(x,s*y,linewidth=1,color=line_color)
            if self.directed:
                xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                yd = [y0[2]-1,y0[2],y0[2]+1]
                yd = [y*s for y in yd]
                ax.plot(xd,yd,linewidth=1,color=line_color)
            if self.weighted:
                xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                yd = [y0[2]-1,y0[2],y0[2]+1]
                yd = [y*s for y in yd]
                ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
    ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
    ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
     bbox=dict(facecolor='w',boxstyle="circle"))
ax.axis('off')
ax.set_aspect(1.0)
plt.show(block = True)
```

graph_AM.py

```
class Graph:

    # Constructor

    def __init__(self, vertices, weighted=False, directed = False):

        self.am = np.zeros((vertices,vertices),dtype=int)-1

        self.weighted = weighted

        self.directed = directed

        self.representation = 'AM'



    # Insert edge function that adds an edge given its source, destination and weight

    def insert_edge(self,source,dest,weight=1):



        # check parameters are valid

        if source >= len(self.am) or dest>=len(self.am) or source <0 or dest<0:

            print('Error, vertex number out of range')

        elif weight!=1 and not self.weighted:

            print('Error, inserting weighted edge to unweighted graph')

        else:

            # check if not directed

            if not self.directed:

                # insert weight to both valid positons for undirected

                self.am[source][dest]=weight

                self.am[dest][source]=weight



                # otherwise, graph is directed and must only be inserted one way
```

```
        else:
            self.am[source][dest]=weight
    return

# Delete edge function that deletes an edge given its source and destination
def delete_edge(self,source,dest):

    # check if parameters are valid
    if source >= len(self.am) or dest>=len(self.am) or source <0 or dest<0:
        print('Error, vertex number out of range')
    else:
        # check if directed or undirected
        if not self.directed:
            # if undirected, delete or subsitute to -1 both positions
            self.am[source][dest]=-1
            self.am[dest][source]=-1
            # otherwise for directed, only one position is deleted
        else:
            self.am[source][dest]=-1
    return

# Display function to print the entire adjacency matrix edges
def display(self):
    print(self.am)
```

```
# Draw function that converts the current graph to an adjacency list then draws
# the graph (as per lab instructions)
def draw(self):

    # converts current graph to an adjacency list to be used for the function
    adjlist = self.as_AL()

    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(adjlist.al)):
        for edge in adjlist.al[i]:
            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color='k')
                if self.directed:
```

```
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color='k')
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
        ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)

# as_EL converts current adjacency matrix graph to an edge list
def as_EL(self):

    # create an empty graph with the same length as current graph
    edgelist=EL.Graph(len(self.am), self.weighted, self.directed)

    # insert edges using a nested loop
    for row in range(len(self.am)):
        for col in range(len(self.am[row])):
            if self.am[row][col]!=-1:
```

```
                edgelist.insert_edge(row,col,self.am[row][col])
    return edgelist

# as_AL converts current adjacency matrix graph to an adjacency list
def as_AL(self):

    # create an empty graph with the same length as current graph
    adjlist=AL.Graph(len(self.am), self.weighted, self.directed)

    # insert edges using a nested loop
    for row in range(len(self.am)):
        for col in range(len(self.am[row])):
            if self.am[row][col]!=-1:
                adjlist.insert_edge(row,col,self.am[row][col])
    return adjlist

def as_AM(self):
    return self

# Breadth first search function used to return path
def BFS(self, s,end):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.am))
```

```
    # Create a queue for BFS
    queue = [[s]]
    while queue:

        # pop element from queue and assign it to s
        s = queue.pop(0)
        # if end is found, return
        if s==end:
            return


        # Get all adjacent vertices of the
        # popped vertex s. If adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in range(len(self.am[s[-1]])):
            if self.am[s[-1]][i] != -1 and visited[i]==False:
                queue.append(s+[i])
                visited[i] = True
    print('From AM BFS')
    return s


# Depth first search function used to return path
def DFS(self, s, end):
```

```
    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From AM DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:

        # check if visited is not empty and if the last element of
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return 'in AM DFS'

        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for i in range(len(self.am[s])):
            if self.am[s][i] != -1:
```

```
                self.DFS_(visited, i, end)
    return visited


# Function to print the path in the correct format as shown in the lab
# instructions [b0,b1,b2,b3]
def path_steps(self, func):
    if func == 'DFS':
        search_path = self.DFS(0,len(self.am)-1)
    if func == 'BFS':
        search_path = self.BFS(0,len(self.am)-1)

    for i in search_path:
        print (i, [int(x) for x in list('{0:04b}'.format(i))])

# Modified draw function from class website used to highlight path
#found from BFS or DFS
def draw_path(self, func):

    if func == 'DFS':
        search_path = self.DFS(0,len(self.am)-1)
    if func == 'BFS':
        search_path = self.BFS(0,len(self.am)-1)

    scale = 30
    fig, ax = plt.subplots()
```

```
# create path list to be used to highlight path
path = []

for j in range(len(search_path)-1):
    path.append((search_path[j], search_path[j+1]))

adjlist=self.as_AL()

for i in range(len(adjlist.al)):
    for j in adjlist.al[i]:
        # highlighted path
        if (i, j.dest) in path or (j.dest, i) in path:
            line_color = "#ff007f"

        else:
            line_color = "#eeefff"

        d,w = j.dest, j.weight
        if self.directed or d>i:
            x = np.linspace(i*scale,d*scale)
            x0 = np.linspace(i*scale,d*scale,num=5)
            diff = np.abs(d-i)
            if diff == 1:
                y0 = [0,0,0,0,0]
```

```
            else:
                y0 = [0,-6*diff,-8*diff,-6*diff,0]
            f = interp1d(x0, y0, kind='cubic')
            y = f(x)
            s = np.sign(i-d)
            ax.plot(x,s*y,linewidth=1,color=line_color)
            if self.directed:
                xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                yd = [y0[2]-1,y0[2],y0[2]+1]
                yd = [y*s for y in yd]
                ax.plot(xd,yd,linewidth=1,color=line_color)
            if self.weighted:
                xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                yd = [y0[2]-1,y0[2],y0[2]+1]
                yd = [y*s for y in yd]
                ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
    ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
    ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
     bbox=dict(facecolor='w',boxstyle="circle"))
ax.axis('off')
ax.set_aspect(1.0)
plt.show(block = True)
```

graph_EL.py

```
# Edge list representation of graphs

import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import interp1d

import graph_AM as AM

import graph_AL as AL

import sys


sys.setrecursionlimit(150000)


class Edge:

    def __init__(self, source, dest, weight=1):

        self.source = source

        self.dest = dest

        self.weight = weight


class Graph:

    # Constructor

    def __init__(self, vertices, weighted=False, directed = False):

        self.el = []

        self.vertices = vertices

        self.weighted = weighted

        self.directed = directed

        self.representation = 'EL'
```

```
# Insert edge function that adds an edge given its source, destination and weight
def insert_edge(self,source,dest,weight=1):

    if weight!=1 and not self.weighted:
        print('Error, inserting weighted edge to unweighted graph')
    else:
        # insert edge to graph
        self.el.append(Edge(source,dest,weight))

# Delete edge function that deletes an edge given its source and destination
def delete_edge(self,source,dest):

    # find position of the given edge and remove from graph
    for i in self.el:
        if i.source==source and i.dest==dest:
            self.el.remove(i)

# Displays all the edges in the graph
def display(self):
    print('[',end='')
    for i in self.el:
        print('('+str(i.source)+','+str(i.dest)+','+str(i.weight)+')',end='')
    print(']',end=' ')
    print('')
```

```
# Draw function that converts the current graph to an adjacency list then draws
# the graph (as per lab instructions)
def draw(self):

    # converts current graph to an adjacency list to be used for the function
    adjlist = self.as_AL()

    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(adjlist.al)):
        for edge in adjlist.al[i]:
            d,w = edge.dest, edge.weight
            if self.directed or d>i:
                x = np.linspace(i*scale,d*scale)
                x0 = np.linspace(i*scale,d*scale,num=5)
                diff = np.abs(d-i)
                if diff == 1:
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color='k')
```

```
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color='k')
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
        ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)


def as_EL(self):
    return self


# as_AM converts current edge list graph to an adjacency matrix
def as_AM(self):


     # create an empty graph with the same length as current graph
    matrix =  AM.Graph(self.vertices, self.weighted, self.directed)
```

```
    # insert edges using a loop
    for i in self.el:
        matrix.insert_edge(i.source, i.dest, i.weight)
    return matrix

# as_AM converts current adjacency matrix graph to an adjacency list
def as_AL(self):

     # create an empty graph with the same length as current graph
    adjlist =  AL.Graph(self.vertices, self.weighted, self.directed)

    # insert edges using a loop
    for i in self.el:
        adjlist.insert_edge(i.source, i.dest, i.weight)
    return adjlist

# Breadth first search function used to return path
def BFS(self, s,end):

    # Mark all the vertices as not visited
    visited = [False] * (self.vertices)
    # Assign visited element at s to True
    visited[s]=True
    # Create a queue for BFS
    queue = [[s]]
```

```
    # Create a path list with s as the first element
    path=[s]

    while queue:

        # pop element from queue and assign it to s
        s = queue.pop(0)
        if s==end:
            return



        # Get all adjacent vertices of the
        # popped vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and append it
        for i in self.el:
            if visited[i.dest] == False:
                queue.append(i.dest)
                visited[i.dest] = True
                path.append(i.dest)
        print('From EL BFS')
        return path

# Depth first search function used to return path
def DFS(self, s, end):
```

```
    # start an empty list of visited elements
    visited=[]

    # call DFS helper function
    print('From EL DFS')
    return self.DFS_(visited, s, end)

# Depth first search helper function used to return path
def DFS_(self, visited, s, end):

    # check if s is in the visited list
    if s not in visited:

        # check if visited is not empty and if the last element of
        # visited is the end element
        if len(visited) > 0 and visited[-1]==end:
            return

        # append s to visited list
        visited.append(s)

        # call function recursively with the starting element as the
        # destination of the neighbours of s
        for neighbour in self.el:
```

```
            self.DFS_(visited, neighbour.dest, end)


    return visited


# Function to print the path in the correct format as shown in the lab
# instructions [b0,b1,b2,b3]
def path_steps(self, func):
    if func == 'DFS':
        search_path = self.DFS(0,self.vertices-1)
    if func == 'BFS':
        search_path = self.BFS(0,self.vertices-1)


    for i in search_path:
        print (i, [int(x) for x in list('{0:04b}'.format(i))])


# Modified draw function from class website used to highlight path
#found from BFS or DFS
def draw_path(self, func):


    if func == 'DFS':
        search_path = self.DFS(0,self.vertices-1)
    if func == 'BFS':
        search_path = self.BFS(0,self.vertices-1)


    scale = 30
```

```
fig, ax = plt.subplots()


# create path list to be used to highlight path

path = []


for j in range(len(search_path)-1):

    path.append((search_path[j], search_path[j+1]))


adjlist=self.as_AL()


for i in range(len(adjlist.al)):

    for j in adjlist.al[i]:

        # highlighted path

        if (i, j.dest) in path or (j.dest, i) in path:

            line_color = "#ff007f"


        else:

            line_color = "#eeefff"


        d,w = j.dest, j.weight

        if self.directed or d>i:

            x = np.linspace(i*scale,d*scale)

            x0 = np.linspace(i*scale,d*scale,num=5)

            diff = np.abs(d-i)

            if diff == 1:
```

```
                    y0 = [0,0,0,0,0]
                else:
                    y0 = [0,-6*diff,-8*diff,-6*diff,0]
                f = interp1d(x0, y0, kind='cubic')
                y = f(x)
                s = np.sign(i-d)
                ax.plot(x,s*y,linewidth=1,color=line_color)
                if self.directed:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.plot(xd,yd,linewidth=1,color=line_color)
                if self.weighted:
                    xd = [x0[2]+2*s,x0[2],x0[2]+2*s]
                    yd = [y0[2]-1,y0[2],y0[2]+1]
                    yd = [y*s for y in yd]
                    ax.text(xd[2]-s*2,yd[2]+3*s, str(w), size=12,ha="center", va="center")
            ax.plot([i*scale,i*scale],[0,0],linewidth=1,color='k')
            ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
             bbox=dict(facecolor='w',boxstyle="circle"))
        ax.axis('off')
        ax.set_aspect(1.0)
        plt.show(block = True)
```

test_graphs.py

```
import matplotlib.pyplot as plt
```

```
import graph_AL as AL_test
import graph_AM as AM_test # Replace line 3 by this one to demonstrate adjacy maxtrix implementation
import graph_EL as EL_test # Replace line 3 by this one to demonstrate edge list implementation


def tests(choice):


    plt.close("all")


    if choice == 1:
        impl=AL_test
        print('\nYou selected Adjacency List\n')
    if choice == 2:
        impl=AM_test
        print('\nYou selected Adjacency Matrix\n')
    if choice == 3:
        impl=EL_test
        print('\nYou selected Edge List\n')


    g = impl.Graph(6)
    g.insert_edge(0,1)
    g.insert_edge(0,2)
    g.insert_edge(1,2)
    g.insert_edge(2,3)
    g.insert_edge(3,4)
    g.insert_edge(4,1)
```

```
g.display()

g.delete_edge(1,2)

g.display()

g.draw()



g = impl.Graph(6,directed = True)

g.insert_edge(0,1)

g.insert_edge(0,2)

g.insert_edge(1,2)

g.insert_edge(2,3)

g.insert_edge(3,4)

g.insert_edge(4,1)

g.display()

g.draw()

g.delete_edge(1,2)

g.display()

g.draw()


g = impl.Graph(6,weighted=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.display()

g.draw()

g.delete_edge(1,2)

g.display()

g.draw()


g = impl.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.display()

g.draw()

g.delete_edge(1,2)

g.display()

g.draw()


print('\nas_AL')

g1=g.as_AL()

g1.draw()

g1.display()
```

```
print('\nas_AM')

g2=g.as_AM()

g2.draw()

g2.display()


print('\nas_EL')

g3=g.as_EL()

g3.draw()

g3.display()
```

Academic Honesty Statement:

“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.”

-Laurence Labayen