**LAB 6**

**DANIEL CRUZ**

**Introduction**

The purpose of this lab is to show how to search within a graph. This lab involves designing the three different implementations of graphs and to write the two different search algorithms for each of these implementations. In this way, the lab will provide insight in the different ways a graph can be represented and how breadth first and depth first search differ.

**Proposed Solution Design and Implementation**

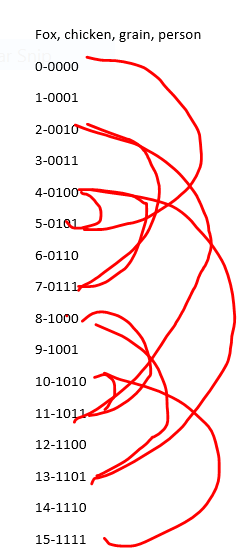
The programs provided on the class website were used as a template to develop the three graph types. In each graph type, three functions were created to convert each graph type to the other graph types. All three of the functions followed the same algorithm, for each edge saved in the graph, call insert\_edge() in order to build the new graph. In some instances, such as adjacency matrix to adjacency list, safeguards needed to be put in place to make sure undirected graphs don’t input the same edge in twice. In this case, the functions check if the graph is directed or not and then only inserts the first diagonal of the matrix to make sure edges are not repeated. In the case of adjacency list to edge list, a list had to be created to keep track of every inserted edge.

The search algorithms in each graph implementation was built spanning five functions. Two functions perform the algorithm itself for breadth first and depth first search. The algorithms were created using the pseudocode from ZyBooks. These functions return a list where each index contains its predecessor leading back to the start specified as a parameter to the functions. There is a function that can take in this path and a destination vertex, and then prints the path to this destination. These two functions can both be called at once using the printBreadthPath(start, end) and printDepthPath(start, end).

In the lab program, a function will be used to create the graph based on the given problem using the adjacency list class. The conversion functions will be used to build the edge list and matrix adjacency implementations and return all three graph types. The paths that each search algorithm calculates for each graph implementations are printed.

For solving the fox, chicken, grain, person problem I sought to bit encode every scenario as recommended in the lab instructions. The following is the bit encodings. Using this, I found which scenarios were valid given the rules and how these valid scenarios can be connected together. These edges were then hand encoded to building the graphs.

Fox, chicken, grain, person

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

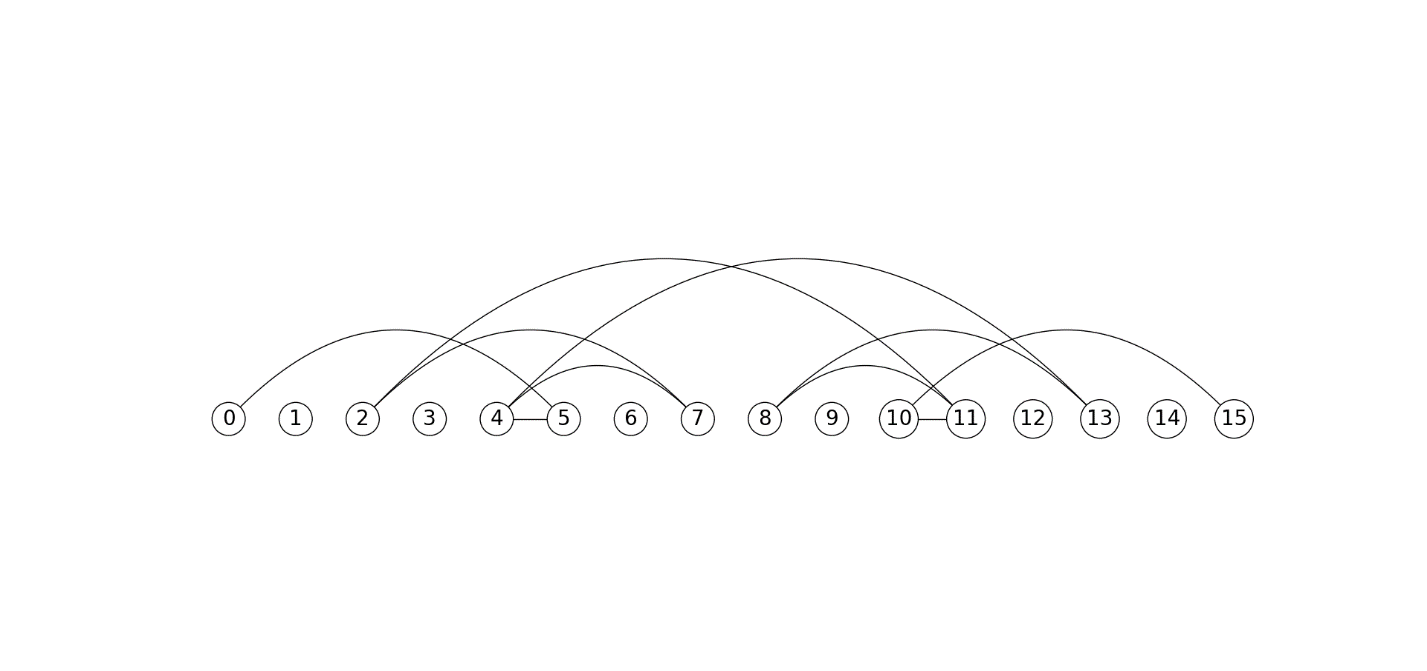
13-1101

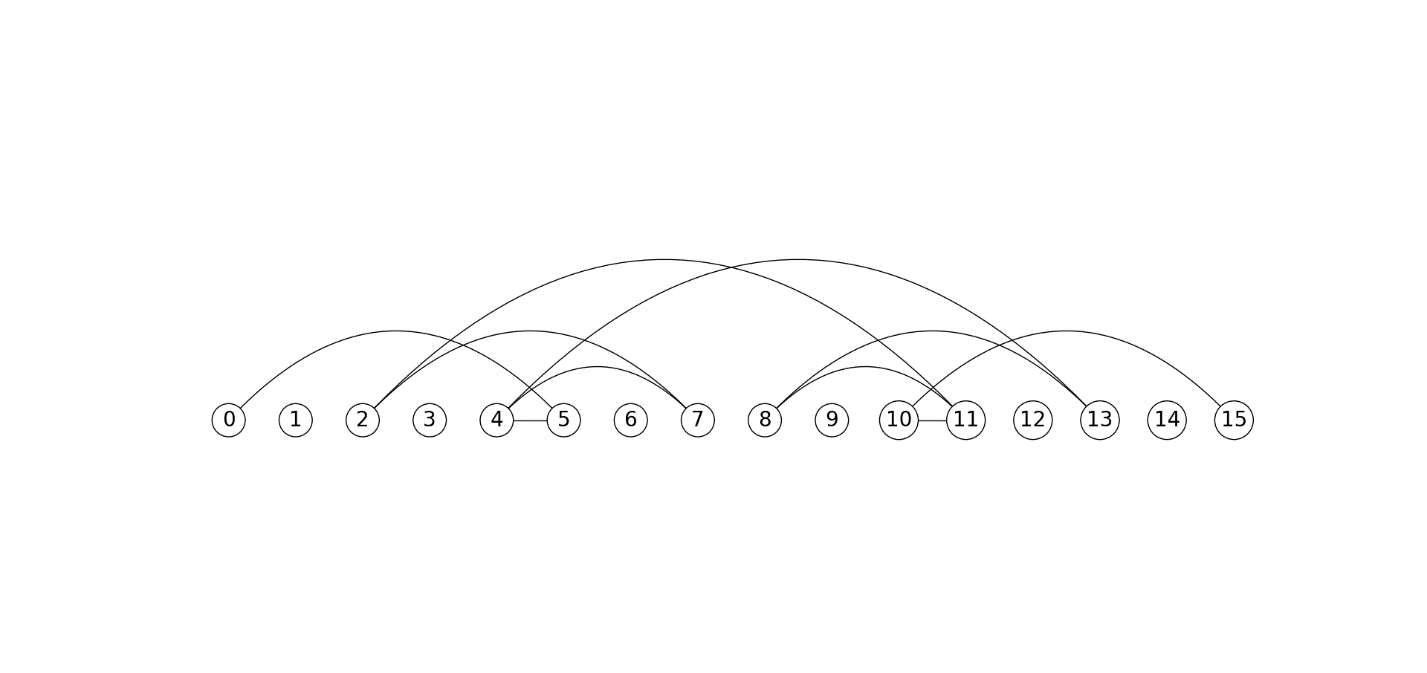
14-1110

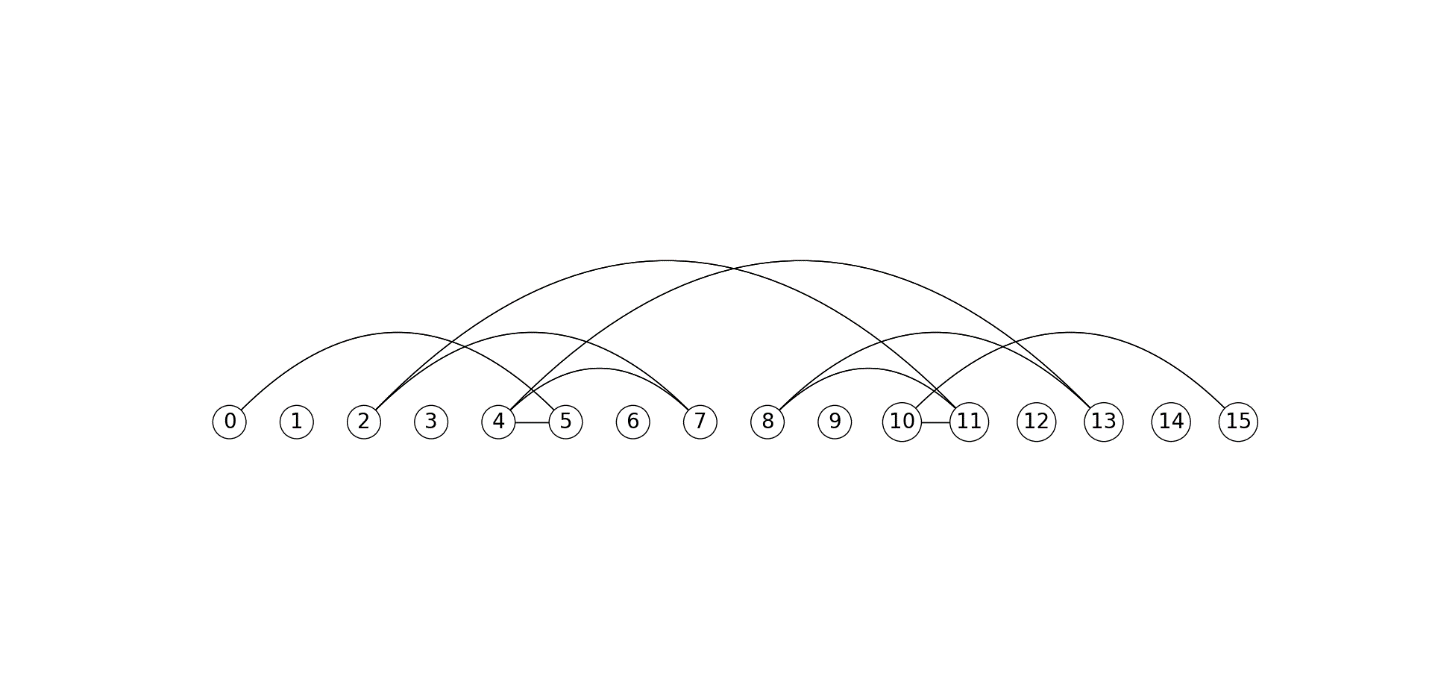
15-1111

**Experimental Results**

The first test here is to call draw() on each graph implementation to see if the graph was built correctly and if all the implementations built the same graph. Figure 1-3 relate to AL, AM, and EL respectively.

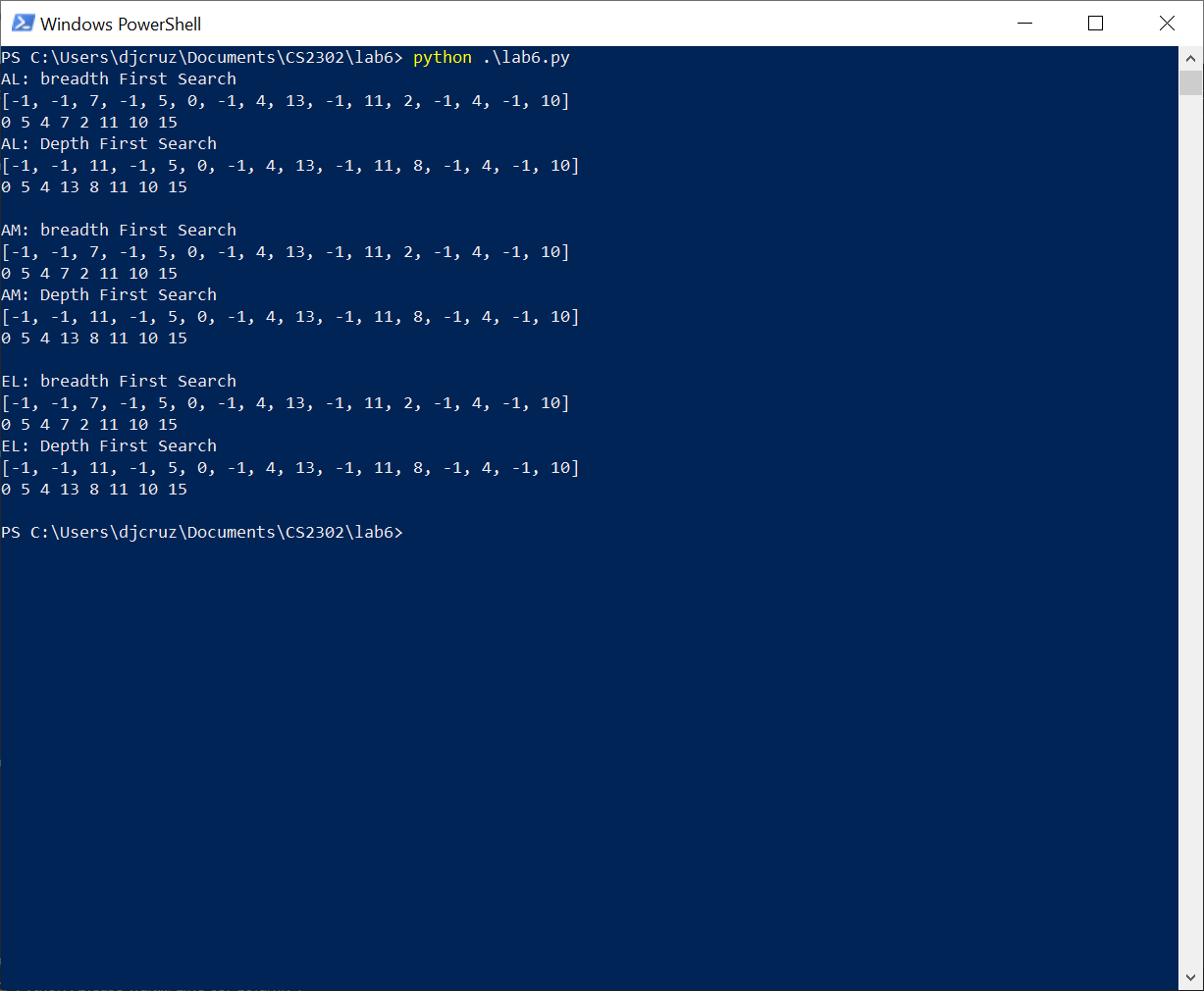






All implementations built the same graph with the same edges.

The next test will run both search algroithsm for every graph implementation and print the path list each search algorithm found. It will then print the quickest path from vertex 0 to 15.



Across all graph implementations, the search algorithms found the same results. This is as expected since all implementations are meant to build the same graph. However, breadth first and depth first search algorithms found different yet valid paths. Manually, I found the solution 0,5,4,13,8,11,10,15 and assumed that was the only possible solution. Breadth first search found a new solution 0,5,4,7,2,11,10,15 that is also valid. Breadth first search is supposed to find the shortest path of an undirected unweighted graph, however both solutions are equal in length so this is not very apparent in this graph. An interesting side effect of building a path using these search algorithms is that this path can be used to navigate from any vertex to 0. In other words, these algorithms did not just find the path from 0 to 15 but from 0 to any other vertex in the graph provided a path exists from 0 to that vertex.

**Conclusion**

Graphs can be implemented using adjacency lists, adjacency matrix, and edge lists. Any graph can either by weighted (meaning the edges have a specific value) and they can be directed (each edge can only be traversed one way). Two ways to traverse a graph is through a breadth first search algorithm and a depth first search algorithm. Both algorithms are very similar, differing in only using a FIFO or LIFO data structure to travel between nodes. However, both algorithms have the possibility of producing unique solutions. Breadth first search produces the optimized path from an unweighted undirected with the least number of vertices visited.

import graph\_AL as ALG

import graph\_AM as AMG

import graph\_EL as ELG

def buildGraphs():

vertices = 16

weighted = False

directed = False

AL = ALG.Graph(vertices, weighted, directed)

AL.insert\_edge(0,5)

AL.insert\_edge(2,11)

AL.insert\_edge(2,7)

AL.insert\_edge(4,5)

AL.insert\_edge(4,7)

AL.insert\_edge(4,13)

AL.insert\_edge(8,11)

AL.insert\_edge(8,13)

AL.insert\_edge(10,11)

AL.insert\_edge(10,15)

EL = AL.as\_EL()

AM = AL.as\_AM()

return AL, AM, EL

def drawGraphs():

AL, AM, EL = buildGraphs()

AL.draw()

AM.draw()

EL.draw()

def main():

AL, AM, EL = buildGraphs()

#AL

#print(AL.buildPath(0))

print("AL: breadth First Search")

print(AL.breadthFirstSearch(0,15))

AL.printBreadthPath(0,15)

print("AL: Depth First Search")

print(AL.depthFirstSearch(0,15))

AL.printDepthPath(0,15)

print()

#AM

#print(AM.buildPath(0))

print("AM: breadth First Search")

print(AM.breadthFirstSearch(0,15))

AM.printBreadthPath(0,15)

print("AM: Depth First Search")

print(AM.depthFirstSearch(0,15))

AM.printDepthPath(0,15)

print()

#EL

#print(EL.buildPath(0))

print("EL: breadth First Search")

print(EL.breadthFirstSearch(0,15))

EL.printBreadthPath(0,15)

print("EL: Depth First Search")

print(EL.depthFirstSearch(0,15))

EL.printDepthPath(0,15)

print()

if \_\_name\_\_ == '\_\_main\_\_':

main()

#drawGraphs()

I [Daniel Cruz] 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