**Lab Report 6: Graphs**

**I. Introduction**

This lab consisted on implementing three types of graphs (Adjacency List, Adjacency Matrix, and Edge List) and completing certain functions to make them work such as inserting an edge, removing an edge and displaying the graph. The second part consisted on solving the fox, chicken and sack of grains crossing a river problem implementing graphs. For this, determining the valid vertices and the valid edges was necessary to code. After storing the right edges, an algorithm to find the path from beginning to end had to be made, and finally one to print the steps and interpret them.

**II. Proposed solution design and implementation**

The first part consisted on completing functions for Adjacency Matrix and Edge List. The first step was understanding how each class stored the edges and therefore build the right edge and place it on the correct position. For the matrix, this was using the source and destination as vertices and assign the value of the weight. If the graph was not directed, source and destination were reversed, and weight was once again assigned. For the Edge List, the order to add did not matter so long as the edge was stored (source, destination, weight) and added to the numpy list.

Deleting was done reaching the vertices with source and destination on the Adjacency Matrix and Assigning -1. For the Edge List, this meant iterating through the list of edges until finding the corresponding source and destination and popping that edge. Displaying the Matrix was done with a simple print, while for the edge list, putting the correct brackets and parentheses on a for loop was necessary.

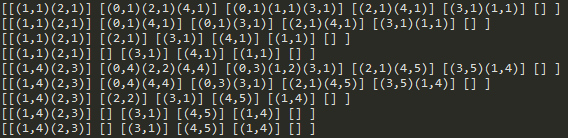
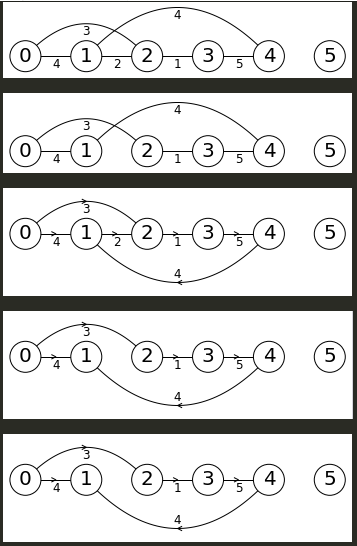
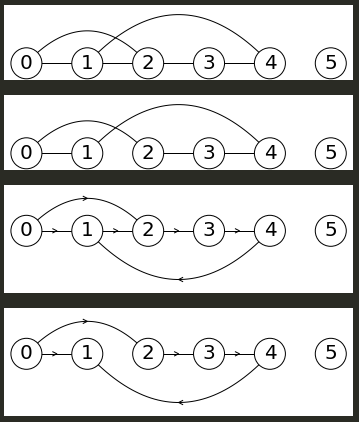
The next step was implementing the other types of lists on the classes. This was done calling the constructor of the desired graph with the correct number of vertices. The correct input for the insert function had to be obtained from each class, then the graph was returned.

The second part was solving the puzzle. My first step was to determine which vertices were possible so that the chicken and the sack of grain weren’t eaten. This was done verifying that the prohibited cases did not happen meaning that bits 3 and 2 were the same and 0 was not, or that bits 2 and 1 were the same and not 0. The available vertices were returned on a set. The second step was verifying the possible edges and inserting them. This was done checking that the last bit was changed because it means the person is moving and that only one other bit could change with it. Comparing the combinations that were on the valid set, they were added as an edge. Modding was used to get the values of these bits.

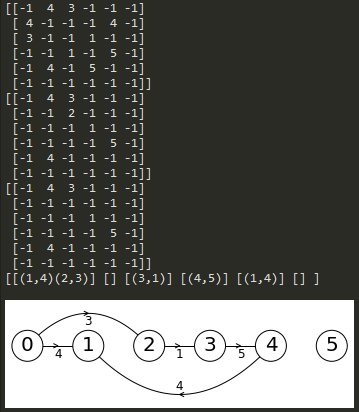
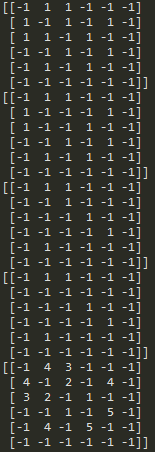
After the graph was formed, a function was made on each graph class that found and returned the path of the graph like the algorithm we saw on class. These algorithms were two: breadth-first search and depth-first search. After obtaining this path, a function returned the path that was needed to get from 0000 to 1111 (0 to 15). This path was returned as a list. Finally, a method was created that translated the instructions from the array into words showing which animal and person moved to which side. This also interpreted the bits from the instructions modding and comparing to the previous instruction. Both algorithms returned 7 steps, but they had a different sequence.

**III. Experimental results**

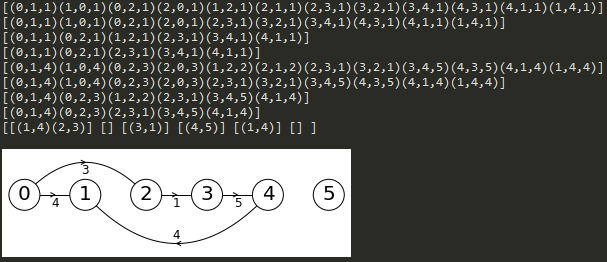
TESTING AL:

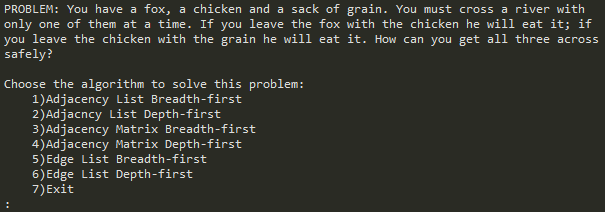
  


TESTING AM:

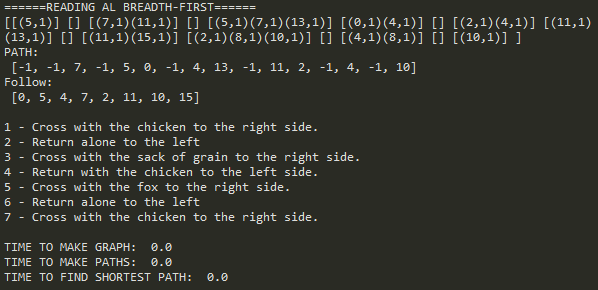


TESTING EL:

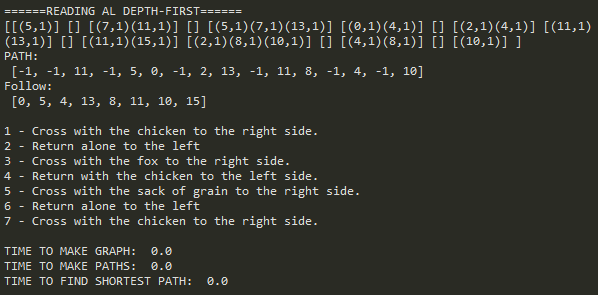


TESTING PART 2:  


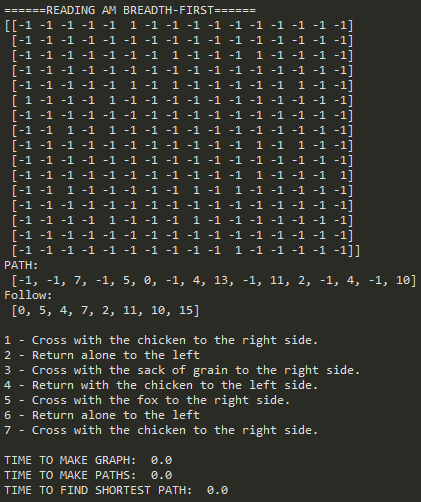
ADJACENCY LIST BREADTH-FIRST:



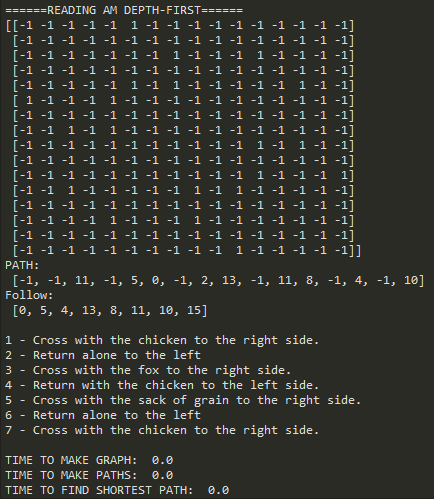
ADJACENCY LIST DEPTH-FIRST:



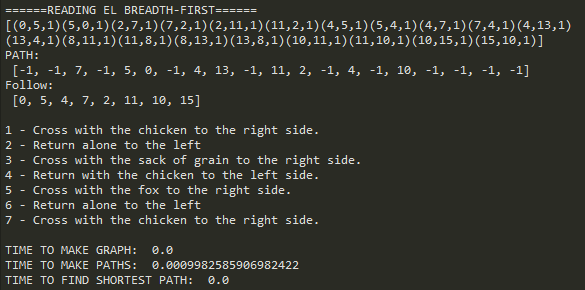
ADJACENCY MATRIX BREADTH-FIRST:



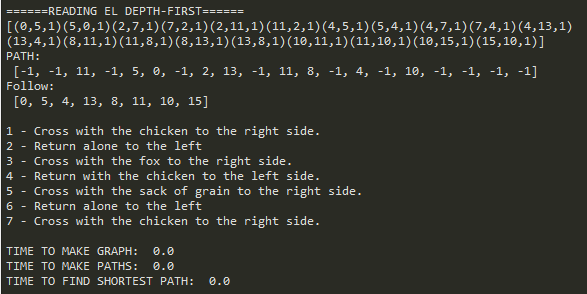
ADJACENCY MATRIX DEPTH-FIRST:



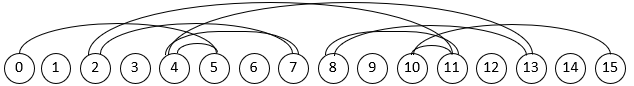
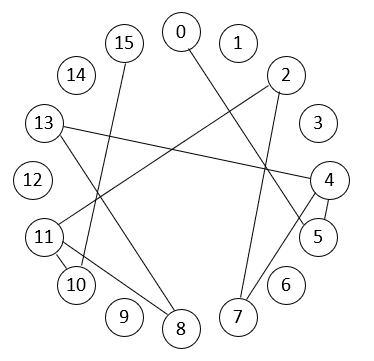
EDGE LIST BREADTH-FIRST:



EDGE LIST DEPTH-FIRST:



DRAWINGS OF GRAPH FROM PART 2:



\*SAME GRAPH

|  |  |  |  |
| --- | --- | --- | --- |
| **GRAPH** | **ADJACENCY LIST** | **ADJACENCY MATRIX** | **EDGE LIST** |
| ADVANTAGES | Easy to traverse  Quick access | Visual  Quickest access  Quick insertion | Quick insertion  Information on vertex |
| DISADVANTAGES | Reserves space for memory even if there is no vertex. | Takes up more memory. | Takes longer to search because it is not sorted. |

**Table:**

**IV. Conclusions**

Graphs are very useful representations of connections and coding them properly can tell you a lot about these connections. Each representation has its own advantage and disadvantages. However, I have seen that the Adjacency List and Matrix implementation of graphs result most effective. At first glance the edge list seemed convenient as it stores the source, destination, and weight, however, this results counter productive when looking for an edge because it requires traversing the list and looking at each edge. For the adjacency List it only requires traversing the vertex, while on the matrix, it requires just getting to the coordinates source and destiny.

This was a very fun lab because it showed that computers really solve logic problems. Sometimes I traced my own results, but when I let the computer do it, it turned out my answer was wrong, and the computer was right. So really, the computer is a tool for us not only for speed in solving problems but also on accuracy. On this lab I learned that data structures are not limited as a way to store data, they can also be a tool to accessing and finding solutions.

**V. Appendix – Source code**

**---Test\_Graph---**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 6 - Test Graph

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/15/2019

Program: Test Graphs

"""

import matplotlib.pyplot as plt

import numpy as np

import graph\_AL as graph

#import graph\_AM as graph # Replace line 3 by this one to demonstrate adjacy maxtrix implementation

#import graph\_EL as graph # Replace line 3 by this one to demonstrate edge list implementation

if \_\_name\_\_ == "\_\_main\_\_":

plt.close("all")

g = graph.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.draw()

g.delete\_edge(1,2)

g.display()

g.draw()

g = graph.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 = graph.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 = graph.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()

g1=g.as\_AL()

g1.display()

g1.draw()

**---Adjacency List---**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 6 - Adjacency List Graph

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/15/2019

Program: implementation of Adjacency List graph.

Inserting edges, deleting edges, displaying the table, breadth-first search,

and depth first search.

"""

# Adjacency list representation of graphs

import numpy as np

import matplotlib.pyplot as plt

import math

from scipy.interpolate import interp1d

import graph\_AM as gam

import graph\_EL as gel

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

self.representation = 'AL'

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')

elif 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 and source!= dest:

self.al[dest].append(Edge(source,weight))

def delete\_edge\_(self,source,dest):

i = 0

for edge in self.al[source]:

if edge.dest == dest:

self.al[source].pop(i) #deletion

return True

i+=1

return False

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')

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(']')

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)

def out\_degree(G,v):

cnt = 0

for i in G[v]:

cnt += 1

return 1

def in\_degree(G,v):

cnt = 0

for i in G:

for j in i:

if j[1] == v:

cnt+= 1

if G.directed:

return cnt

return cnt/2

def as\_EL(self):

#constructor for EL graph

g = gel.Graph(len(self.al), weighted=self.weighted,directed = self.directed)

for i in range(len(self.al)): #iterates through np

for j in range(len(self.al[i])): #list inside np

g.insert\_edge(i,self.al[i][j].dest, self.al[i][j].weight)#inserts corresponding values

return g

def as\_AM(self):

#constructor for AM graph

g = gam.Graph(len(self.al), weighted=self.weighted,directed = self.directed)

for i in range(len(self.al)): #iterates through np array

for j in range(len(self.al[i])): #lists inside np array

g.insert\_edge(i,self.al[i][j].dest, self.al[i][j].weight) #inserts corresponding values

return g

def as\_AL(self):

return self #self already al

def depth\_search(self, start=0):

stack = [start] #Push startV to stack

path = [-1] \* len(self.al)

visited = set()

while (len(stack)>0):

currentV = stack.pop()

if (not currentV in visited):

visited.add(currentV)#Add currentV to visited

for i in self.al[currentV]:# for each vertex adjacent to currentV

stack.append(i.dest)

if (not i.dest in visited):

path[i.dest] = currentV #values on path

return path

def breadth\_search(self, start=0):

fqueue = [start] #Push start to fqueue

path = [-1] \* len(self.al) #path to return

discovered = {start} #Add start to discovered

while (len(fqueue)>0):

currentV = fqueue.pop()

#for each vertex adjacent to currentV

for i in self.al[currentV]:

if (not i.dest in discovered): #if adj not waiting

fqueue.insert(0,i.dest) #follow adj

discovered.add(i.dest) #add queue

path[i.dest]=currentV #add path

return path

**---Adjacency Matrix---**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 6 - Adjacency Matrix Graph

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/15/2019

Program: implementation of Adjacency Matrix graph.

Inserting edges, deleting edges, displaying the table, breadth-first search,

and depth first search.

"""

# Adjacency matrix representation of graphs

import numpy as np

import matplotlib.pyplot as plt

import math

from scipy.interpolate import interp1d

import graph\_AL as gal

import graph\_EL as gel

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'

def insert\_edge(self,source,dest,weight=1):

self.am[source,dest]=weight #change value of corresponding place to the weight

if not self.directed and source!=dest: #double if not directed

self.am[dest,source]=weight

def delete\_edge(self,source,dest):

self.am[source,dest]=-1 #reset corresponding value to -1

if not self.directed: #double if not directed

self.am[dest,source]=-1

def display(self):

print(self.am) #prints table

def draw(self):

return

def as\_EL(self):

#constructor for edge graph

g = gel.Graph(len(self.am), weighted=self.weighted,directed = self.directed)

for source in range(len(self.am)): #iterates length of matrix

for dest in range(len(self.am)):#iterates for width of matrix

if(self.am[source,dest] != -1): #sends only values existing

g.insert\_edge(source, dest, self.am[source,dest]) #inserts corresponding values

return g

def as\_AM(self):

return self #self already AM

def as\_AL(self):

#constructor for AL graph

g = gal.Graph(len(self.am), weighted=self.weighted,directed = self.directed)

for source in range(len(self.am)): #itereates length

for dest in range(len(self.am)): #iterates width

if(self.am[source,dest] != -1): #only values existing

g.insert\_edge(source, dest, self.am[source,dest]) #insert corresponding values

return g

def depth\_search(self, start=0):

stack = [start] #Push startV to stack

path = [-1] \* len(self.am)

visited = set()

while (len(stack)>0):

currentV = stack.pop()

if (not currentV in visited):

visited.add(currentV)#Add currentV to visited

for i in range(len(self.am)):# for each vertex adjacent to currentV

if self.am[currentV,i] != -1:

stack.append(i)

if (not i in visited):

path[i] = currentV

return path

def breadth\_search(self, start=0):

fqueue = [start] #Push start to fqueue

path = [-1] \* len(self.am) #path to return

discovered = {start} #Add start to discovered

while (len(fqueue)>0):

currentV = fqueue.pop()

#for each vertex adjacent to currentV

for i in range(len(self.am)):

if (self.am[currentV, i] != -1 and not i in discovered):

fqueue.insert(0,i)

discovered.add(i)

path[i]=currentV

return path

**---Edge List---**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 6 - Edge List Graph

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/15/2019

Program: implementation of Edge List graph.

Inserting edges, deleting edges, displaying the table, breadth-first search,

and depth first search.

"""

# Edge list representation of graphs

import numpy as np

import matplotlib.pyplot as plt

import math

from scipy.interpolate import interp1d

import graph\_AL as gal

import graph\_AM as gam

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.vertices = vertices

self.el = []

self.weighted = weighted

self.directed = directed

self.representation = 'EL'

def insert\_edge(self,source,dest,weight=1):

self.el.append(Edge(source, dest, weight)) #insert edge with all info

if not self.directed and source!=dest: #double if not directed

self.el.append(Edge(dest, source, weight))

def delete\_edge(self,source,dest):

i = 0

for edge in self.el: #find edges

if edge.source == source and edge.dest == dest: #if edge and dest correspond, this is the edge we are looking for

self.el.pop(i) #remove edge

break;

i+=1

if not self.directed:#double if not directed

i=0

for edge in self.el:

if edge.source == dest and edge.dest == source: #destiny as source

self.el.pop(i)

return True

i+=1

return False

def display(self):

print('[',end='') #all inside brackets

for i in self.el:

print('(' + str(i.source) + ','+ str(i.dest) + ',' + str(i.weight) + ')',end='') #each edge inside parentheses

print(']')

return

def draw(self):

return

def as\_EL(self):

return self

def as\_AM(self):

#constructor for AM

g = gam.Graph(self.vertices, weighted=self.weighted,directed = self.directed)

for edge in self.el: #iterate through edges list

g.insert\_edge(edge.source, edge.dest, edge.weight) #insert corresponding values

return g

def as\_AL(self):

#constructor fot AL

g = gal.Graph(self.vertices, weighted=self.weighted,directed = self.directed)

for edge in self.el: #iterate through edges list

g.insert\_edge(edge.source, edge.dest, edge.weight) #insert corresponding values

return g

def depth\_search(self, start=0):

stack = [start] #Push startV to stack

path = [-1] \* len(self.el)

visited = set()

while (len(stack)>0):

currentV = stack.pop()

if (not currentV in visited):

visited.add(currentV)#Add currentV to visited

for i in self.el:# for each vertex adjacent to currentV

if i.source == currentV:

stack.append(i.dest)

if (not i.dest in visited):

path[i.dest] = currentV

return path

def breadth\_search(self, start=0):

fqueue = [start] #Push start to fqueue

path = [-1] \* len(self.el) #builds path

discovered = {start} #Add start to discovered

while (len(fqueue)>0):

currentV = fqueue.pop()

#for each vertex adjacent to currentV

for i in self.el:

if i.source == currentV:

if (not i.dest in discovered):

fqueue.insert(0,i.dest)

discovered.add(i.dest)

path[i.dest]=currentV

return path

**---Lab 6---**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 6 - Graphs

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 11/15/2019

Program: Solves the fox, chicken, grains crossing the river problem. Implementing

graphs on three versions of it with two search algorithms (breadth first and depth first)

This program computes the valid vertices that the problem can use in function validV().

Fills the graph with the possible edges in fillGraph().

Finds the path to traverse the graph (solve the problem) in findPath()

Interprets this path into words (instructions) in function interpret()

Uses breadth-first and depth-first search algorithms found in each graph's class

Shows running times.

"""

import graph\_AL as gal

import graph\_EL as gel

import graph\_AM as gam

import time

#finds valid vertices for the problem

def validV():

valid = set()

for i in range(16): #16 possible vertices

#if chicken and fox arent left together alone or chicken and grain arent left together alone

if not(((i//8)%2 == (i//4)%2 and (i//4)%2 != i%2) or ((i//4)%2 == (i//2)%2 and (i//4)%2 != i%2)):

valid.add(i) #adds valid vertices

return valid

def fillGraph(g, v):

for i in v:

if i%2 == 0: #person moves from right to left (fills this vertices)

if i+1 in v: #checks if moving the person alone is possible

g.insert\_edge(i,i+1)

if (i//2)%2 == 0 and i+3 in v: #checks if moving with the sack is possible

g.insert\_edge(i,i+3)

if (i//4)%2 == 0 and i+5 in v: #checks if moving with the chicken is possible

g.insert\_edge(i,i+5)

if (i//8)%2 == 0 and i+9 in v: #checks if moving with the fox is possible

g.insert\_edge(i,i+9)

def findPath(path, end):

if path[end] < 0: #arrived to beginning

return [end]

return findPath(path, path[end]) + [end] #follow next index on path

def interpret(way):

cnt = 1

for i in range(1,len(way)):

if way[i]%2 != 0: #means going to the right

if (way[i]//2)%2 != 0 and (way[i-1]//2)%2 == 0: #sack of grain moved

print(cnt, "- Cross with the sack of grain to the right side.")

elif (way[i]//4)%2 != 0 and (way[i-1]//4)%2 == 0: #chicken moved

print(cnt, "- Cross with the chicken to the right side.")

elif (way[i]//8)%2 != 0 and (way[i-1]//8)%2 == 0: #fox moved

print(cnt, "- Cross with the fox to the right side.")

else:

print(cnt, "- Cross alone to the right") #only the person moves

else: #goig to the left

if (way[i]//2)%2 == 0 and (way[i-1]//2)%2 != 0: #sack of grain moved

print(cnt, "- Return with the sack of grain to the left side.")

elif (way[i]//4)%2 == 0 and (way[i-1]//4)%2 != 0: #chicken moved

print(cnt, "- Return with the chicken to the left side.")

elif (way[i]//8)%2 == 0 and (way[i-1]//8)%2 != 0: #fox moved

print(cnt, "- Return with the fox to the left side.")

else:

print(cnt, "- Return alone to the left") #only the person moves

cnt += 1

stay = True #variable to stay on the menu

while(stay):

print("\nPROBLEM: You have a fox, a chicken and a sack of grain. You must cross a river with only one of them at a time. If you leave the fox with the chicken he will eat it; if you leave the chicken with the grain he will eat it. How can you get all three across safely?" )

option = input("Choose the algorithm to solve this problem:\n\t1)Adjacency List Breadth-first\n\t2)Adjacncy List Depth-first\n\t3)Adjacency Matrix Breadth-first\n\t4)Adjacency Matrix Depth-first\n\t5)Edge List Breadth-first\n\t6)Edge List Depth-first\n\t7)Exit\n:")

#fox, chicken, grain, person

#invalid: [1,1,0,0](12) and [0,0,1,1](3)

# [0,1,1,0](6) and [1,0,0,1](9)

# [0,0,0,1](1) and [1,1,1,0] (14)

valid = validV() #valid inputs

bfirst = {"1","3","5"} #options for breadth first

dfirst = {"2","4","6"} #options for depth first

if (option in bfirst or option in dfirst):

if (option in bfirst):

if (option=="1"): #AL Breadth-first

print("======READING AL BREADTH-FIRST======")

g = gal.Graph(16)

elif (option=="3"): #AM Breadth-first

print("======READING AM BREADTH-FIRST======")

g = gam.Graph(16)

else: #EL Breadth-first

print("======READING EL BREADTH-FIRST======")

g = gel.Graph(16)

start = time.time()

fillGraph(g, valid) #call to fill graph that was chosen

timeGraph = time.time()-start

g.display() #show graph

start = time.time()

path = g.breadth\_search() #call to search

timePath = time.time()-start

else:

if (option=="2"): #AL Depth-first

print("======READING AL DEPTH-FIRST======")

g = gal.Graph(16)

elif (option=="4"): #AM Depth-first

print("======READING AM DEPTH-FIRST======")

g = gam.Graph(16)

else: #EL Depth-first

print("======READING EL DEPTH-FIRST======")

g = gel.Graph(16)

start = time.time()

fillGraph(g, valid) #call to fill graph that was chosen

timeGraph = time.time()-start

g.display() #show graph

start = time.time()

path = g.depth\_search() #call to search

timePath = time.time()-start

print("PATH:\n", path)

start = time.time()

connect = findPath(path,15) #call to find the path from 0 to 15

timeFind = time.time()-start

print("Follow:\n", connect,"\n")

interpret(connect)

#show results

print("\nTIME TO MAKE GRAPH: ", timeGraph)

print("TIME TO MAKE PATHS: ", timePath)

print("TIME TO FIND SHORTEST PATH: ", timeFind)

#exit loop

elif (option=="7"):

stay = False

else:

print("Please choose a number between 1 and 7.")

**VI – Academic Honesty Certification**

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.

x- Elisa Jimenez Todd