**CS 2302 Data Structures**

**Spring 2019**

**Lab Report #6**

Due: November 15, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were asked to solve a problem using different representations of graphs. The solution to this problem was the same through all implementations. The difference was found in the implementation of the different representations. These representations are adjacency list, adjacency matrix, and edge list. From these representations we were asked to implement two graph search algorithms. These two graph search algorithms are breadth-first search and depth-first search. Through using these techniques, we can find the solutions to the following prompt:

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? For part 2, you will implement a solution to this problem using a graph search algorithm.

**Proposed Solution Design and Implementation**

**Part #1**

The insert edge, delete edge, and display function for adjacency list were provided by the instructor. For adjacency matrix as well as edge list the display function remains the same since it iterates through the elements of the graph the same and prints the same as well. For adjacency matrix the information is stored within the given source and destination by using the given weight. Empty vertices in the graph are represented with a 0. This same process applies for delete edge function. Within the given source and destination, the weight is set to 0 thus showing that no edge exists between given source and destination. The running time for insert edge and delete edge is constant since a source and destination is given and required. These parameters allow for direct access and verification to insert or delete an edge. The insert edge function for edge list simply appends an edge while the delete edge function first finds the edge that is to be removed and then removes it. After this all remaining elements are shifted to the correct position. The running time for insert edge in edge list is constant since you are simply appending to the list. The delete edge in edge list has a running time of O(E) with E standing for the number of edges in the graph. The display function for adjacency matrix is O(V\*V) with V standing for the number of vertices. This is due to the need to iterate through all the elements of the matrix. The display function for edge list has a running time of O(E) due to the list being composed of the number of edges in the graph and the display function iterating through these values.

**Part #2**

For part 2 we needed to implement two search algorithms. These algorithms were breadth-first search and depth-first search. Breadth-first search uses a queue to search while the depth-first search algorithm uses a stack. The function of a stack and a queue is crucial to the understanding of these search algorithms. The implementation of these search algorithms is the same through all three representations of the graph. To solve the prompt, we needed to find solutions on paper before being able to implement on the graph. The lab assignment suggests representing the state of the world in four bits with each bit representing the location of the fox, chicken, grain, and yourself. After doing this procedure by hand you see that the edges are [(0,10),(10,2),(2,14),(2,11)(14,4),(11,1),(4,13),(1,13),(13,5),(5,15).

From this we can substitute this data into the graph and solve the problem. The solution is found by providing the path of the breadth-first and depth-first search algorithms.

**Experimental Results**

**Using adjacency list**

**﻿[[(10,1)] [(11,1)(13,1)] [(10,1)(14,1)(11,1)] [] [(14,1)(13,1)] [(13,1)(15,1)] [] [] [] [] [(0,1)(2,1)] [(2,1)(1,1)] [] [(4,1)(1,1)(5,1)] [(2,1)(4,1)] [(5,1)] ]**

**Breadth-first search:**

**[0, 10, 2, 11, 1, 13, 5, 15]**

**Depth-first search:**

**[0, 10, 2, 14, 4, 13, 5, 15]**

**Using edge list**

**﻿[[(10,1)] [(11,1)(13,1)] [(10,1)(14,1)(11,1)] [] [(14,1)(13,1)] [(13,1)(15,1)] [] [] [] [] [(0,1)(2,1)] [(2,1)(1,1)] [] [(4,1)(1,1)(5,1)] [(2,1)(4,1)] [(5,1)] ]**

**Breadth-first search:**

**[0, 10, 2, 11, 1, 13, 5, 15]**

**Depth-first search:**

**[0, 10, 2, 14, 4, 13, 5, 15]**

**﻿﻿﻿**

**Using adjacency matrix**

**﻿[[(10,1)] [(11,1)(13,1)] [(10,1)(14,1)(11,1)] [] [(14,1)(13,1)] [(13,1)(15,1)] [] [] [] [] [(0,1)(2,1)] [(2,1)(1,1)] [] [(4,1)(1,1)(5,1)] [(2,1)(4,1)] [(5,1)] ]**

**Breadth-first search:**

**[0, 10, 2, 14, 4, 13, 5, 15]**

**Depth-first search:**

**[0, 10, 2, 11, 1, 13, 5, 15]**

**Conclusion**

In conclusion this lab teaches us how different graphs differ in functionality when trying to solve a common problem. This problem showed us that the same answer is provided by each implementation. The difference is found in how the data is represented and how each representation produces and finds the solution. Adjacency list, adjacency matrix, and edge list all produced a graph as well as depth-first and breadth-first search produced the desired result. Now it is up to the programmer to evaluate which data structure may be the best equipped to solve the problem they are facing. This lab allowed us to see how a data structure can solve a problem in a matter of seconds when we could take a couple of minutes to solve it on paper.

**Appendix**

**﻿**

**Graph\_AL**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import math**

**from scipy.interpolate import interp1d**

**import graph\_AM**

**import graph\_EL**

**from collections import deque**

**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:**

**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)**

**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 as\_EL(self):**

**g = graph\_EL.Graph(len(self.al), weighted=self.weighted, directed=self.directed)**

**for i in range(len(self.al)):**

**for edge in self.al[i]:**

**if not self.directed:**

**if i < edge.dest:**

**g.insert\_edge(i, edge.dest, edge.weight)**

**else:**

**g.insert\_edge(i, edge.dest, edge.weight)**

**return g**

**def as\_AM(self):**

**g = graph\_AM.Graph(len(self.al), weighted=self.weighted, directed=self.directed)**

**for i in range(len(self.al)):**

**for edge in self.al[i]:**

**if not self.directed:**

**if i < edge.dest:**

**g.insert\_edge(i, edge.dest, edge.weight)**

**else:**

**g.insert\_edge(i, edge.dest, edge.weight)**

**return g**

**def as\_AL(self):**

**return self**

**def dfsR(self, begin, end, visited):**

**if begin == end:**

**return [end]**

**tempVisited = list(visited)**

**tempVisited.append(begin)**

**for edge in self.al[begin]:**

**if edge.dest not in tempVisited:**

**path = self.dfsR(edge.dest, end, tempVisited)**

**if path is not None:**

**path.insert(0, begin)**

**return path**

**# depth-first search**

**def dfs(self, begin, end):**

**path = self.dfsR(begin,end, [])**

**if path is None:**

**return []**

**return path**

**# breadth - first search**

**def bfs(self, begin, end):**

**q = deque()**

**q.append(begin)**

**visited = []**

**parent = [-1 for i in range(len(self.al))]**

**found = False**

**while not found and len(q) != 0:**

**temp = q.popleft()**

**visited.append(temp)**

**for edge in self.al[temp]:**

**if edge.dest not in visited:**

**parent[edge.dest] = temp**

**if edge.dest == end:**

**found = True**

**break**

**q.append(edge.dest)**

**path = []**

**if not found:**

**return path**

**temp = end**

**while temp != begin:**

**path.insert(0, temp)**

**temp = parent[temp]**

**path.insert(0, begin)**

**return path**

**﻿graph\_AM**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import math**

**from scipy.interpolate import interp1d**

**import graph\_AL**

**import graph\_EL**

**from collections import deque**

**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):**

**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:**

**self.am[source][dest] = weight**

**if not self.directed:**

**self.am[dest][source] = weight**

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

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

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

**else:**

**if self.am[source][dest] == 0:**

**print('Error, edge to delete not found')**

**self.am[source][dest] = 0**

**if not self.directed:**

**self.am[dest][source] = 0**

**def display(self):**

**print('[', end='')**

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

**print('[', end='')**

**for j in range(len(self.am[i])):**

**if self.am[i][j] != 0:**

**print('(' + str(j) + ',' + str(self.am[i][j]) + ')', end='')**

**print(']', end=' ')**

**print(']')**

**def draw(self):**

**g = self.as\_AL()**

**g.draw()**

**def as\_EL(self):**

**g = graph\_EL.Graph(len(self.am), weighted=self.weighted, directed=self.directed)**

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

**for j in range(len(self.am[i])):**

**if self.am[i][j] != 0:**

**if not self.directed:**

**if i < j:**

**g.insert\_edge(i, j, self.am[i][j])**

**else:**

**g.insert\_edge(i, j, self.am[i][j])**

**return g**

**def as\_AM(self):**

**return self**

**def as\_AL(self):**

**g = graph\_AL.Graph(len(self.am), weighted=self.weighted, directed=self.directed)**

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

**for j in range(len(self.am[i])):**

**if self.am[i][j] != 0:**

**if not self.directed:**

**if i < j: # prevent double edges**

**g.insert\_edge(i, j, self.am[i][j])**

**else:**

**g.insert\_edge(i, j, self.am[i][j])**

**return g**

**# depth-first search recursive**

**def dfsR(self, begin, end, visited):**

**if begin == end:**

**return [end]**

**tempVisited = list(visited)**

**tempVisited.append(begin)**

**for j in range(len(self.am[begin])):**

**if self.am[begin][j] != 0:**

**if j not in tempVisited:**

**path = self.dfsR(j, end, tempVisited)**

**if path is not None:**

**path.insert(0, begin)**

**return path**

**# depth-first search**

**def dfs(self, begin, end):**

**path = self.dfsR(begin, end, [])**

**if path is None:**

**return []**

**return path**

**# breadth - first search**

**def bfs(self, begin, end):**

**q = deque()**

**q.append(begin)**

**visited = []**

**parent = [0 for i in range(len(self.am))]**

**found = False**

**while not found and len(q) != 0:**

**temp = q.popleft()**

**visited.append(temp)**

**for j in range(len(self.am[temp])):**

**if self.am[temp][j] != 0:**

**if j not in visited:**

**parent[j] = temp**

**if j == end:**

**found = True**

**break**

**q.append(j)**

**path = []**

**if not found:**

**return path**

**temp = end**

**while temp != begin:**

**path.insert(0, temp)**

**temp = parent[temp]**

**path.insert(0, begin)**

**return path**

**﻿graph\_EL**

**import numpy as np**

**import matplotlib.pyplot as plt**

**import math**

**from scipy.interpolate import interp1d**

**import graph\_AL**

**import graph\_AM**

**from collections import deque**

**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):**

**if source >= self.vertices or dest >= self.vertices 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.el.append(Edge(source, dest, weight))**

**if not self.directed:**

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

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

**if source >= self.vertices or dest >= self.vertices or source < 0 or dest < 0:**

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

**else:**

**index = -1**

**for i in range(len(self.el)):**

**if self.el[i].source == source and self.el[i].dest == dest:**

**index = i**

**if index != -1:**

**del self.el[index]**

**if not self.directed:**

**index = -1**

**for i in range(len(self.el)):**

**if self.el[i].source == dest and self.el[i].dest == source:**

**index = i**

**if index != -1:**

**del self.el[index]**

**else:**

**print('Error, edge to delete not found')**

**def display(self):**

**print('[', end='')**

**for i in range(self.vertices):**

**print('[', end='')**

**for j in self.el:**

**if j.source == i:**

**print('(' + str(j.dest) + ',' + str(j.weight) + ')', end='')**

**print(']', end=' ')**

**print(']')**

**def draw(self):**

**g = self.as\_AL()**

**g.draw()**

**def as\_EL(self):**

**return self**

**def as\_AM(self):**

**g = graph\_AM.Graph(self.vertices, weighted=self.weighted, directed=self.directed)**

**for i in range(self.vertices):**

**for j in self.el:**

**if j.source == i:**

**if not self.directed:**

**if i < j.dest:**

**g.insert\_edge(i, j.dest, j.weight)**

**else:**

**g.insert\_edge(i, j.dest, j.weight)**

**return g**

**def as\_AL(self):**

**g = graph\_AL.Graph(self.vertices, weighted=self.weighted, directed=self.directed)**

**for i in range(self.vertices):**

**for j in self.el:**

**if j.source == i:**

**if not self.directed:**

**if i < j.dest: # prevent double edges**

**g.insert\_edge(i, j.dest, j.weight)**

**else:**

**g.insert\_edge(i, j.dest, j.weight)**

**return g**

**# depth-first search recursive**

**def dfsR(self, begin, end, visited):**

**if begin == end:**

**return [end]**

**tempVisited = list(visited)**

**tempVisited.append(begin)**

**for j in self.el:**

**if j.source == begin:**

**if j.dest not in tempVisited:**

**path = self.dfsR(j.dest, end, tempVisited)**

**if path is not None:**

**path.insert(0, begin)**

**return path**

**def dfs(self, begin, end):**

**path = self.dfsR(begin, end, [])**

**if path is None:**

**return []**

**return path**

**# breadth - first search**

**def bfs(self, begin, end):**

**q = deque()**

**q.append(begin)**

**visited = []**

**parent = [-1 for i in range(self.vertices)]**

**found = False**

**while not found and len(q) != 0:**

**temp = q.popleft()**

**visited.append(temp)**

**for j in self.el:**

**if j.source == temp:**

**if j.dest not in visited:**

**parent[j.dest] = temp**

**if j.dest == end:**

**found = True**

**break**

**q.append(j.dest)**

**path = []**

**if not found:**

**return path**

**temp = end**

**while temp != begin:**

**path.insert(0, temp)**

**temp = parent[temp]**

**path.insert(0, begin)**

**return path**

**﻿MAIN**

**﻿import matplotlib.pyplot as plt**

**import graph\_AM as graph**

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

**plt.close("all")**

**g = graph.Graph(16)**

**g.insert\_edge(0, 10)**

**g.insert\_edge(10, 2)**

**g.insert\_edge(2, 14)**

**g.insert\_edge(2, 11)**

**g.insert\_edge(14, 4)**

**g.insert\_edge(11, 1)**

**g.insert\_edge(4, 13)**

**g.insert\_edge(1, 13)**

**g.insert\_edge(13, 5)**

**g.insert\_edge(5, 15)**

**g.display()**

**g.draw()**

**print("Breadth-first search:")**

**print(g.bfs(0, 15))**

**print("Depth-first search:")**

**print(g.dfs(0, 15))**

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