**Lab Report 7: Algorythms**

**I. Introduction**

There were two problems to solve in this lab. The first was to check if a graph had a Hamiltonian cycle, and this was to be implemented in two ways: randomization and backtracking. The program would let the user test the default graph, create a new graph, or generate a random graph. It returns the edges connected that make the hamiltonian cycle or none if none was found. It also shows the comparisons made and the time taken to complete. Randomization would be achieved taking random edges and see if that made the Hamiltonian path. Backtracking would be solved testing possible solutions until one was found or all the possibilities were done.

The second problem was to implement the edit distance algorythm with the exception that only replacements between vowels or between consonants are allowed. This method would print the edit distance, the size of the matrix, and the time.

**II. Proposed solution design and implementation**

The first thing to solve was figuring out what the method took as input. I decided on the number of vertices on the graph, and the list of edges of the graph. Therefore, an edge list was to be implemented.

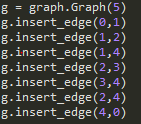
For randomization, I needed a loop that iterated a considerable amount of times, not too many but not too little so that the method would be as efficient as possible. I decided on 2 to the number of edges. Inside the loop, I needed to test the graphs with random chosen edges, so I created a helper method that took a random set of edges of the graph and returned the graph. Then a check was made to see if that graph was Hamiltonian, if it was, it returned the list of edges and printed them, if not, the loop continued, and if the loop was over none was returned.

For backtracking I based my code on the sum subset code given in class. I adapted it to have as base case when the graph was Hamiltonian or the list of edges was over. The method takes the list of edges and adds to the graph until there are the same edges as number of vertices, if the graph formed was not Hamiltonian, then backtracking took place and another edge was to be taken. If no graph met the requirements, None was returned.

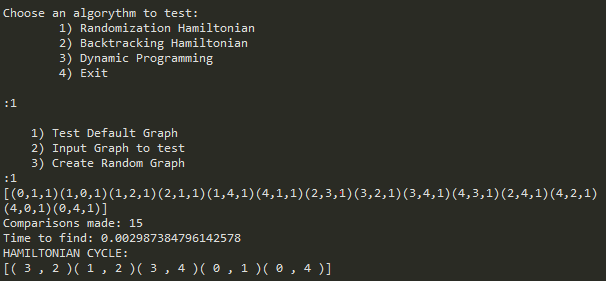
For the edit distance, a simple change was made. A set of vowels was added to be compared, if there ever was the case a replacement had to be made (that happened when the top left number was taken to fill the matrix) then a check was made that either both were vowels or none were, and if this condition was met, then a replacement was not allowed and either a deletion or insertion took place, so the minimum value of the top or left was taken and added 1.

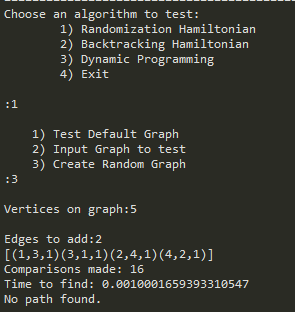
**III. Experimental results**

Default graph is:

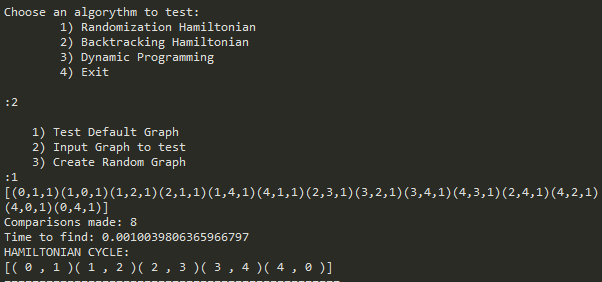
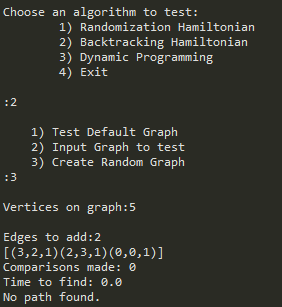
 Which does have a Hamiltonian cycle 0-1-2-3-4-0

Testing randomization:

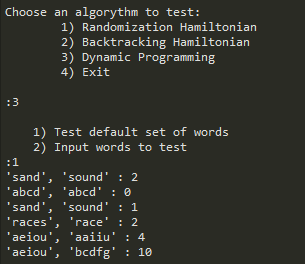


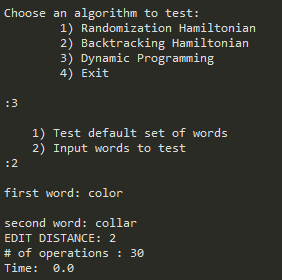


Testing Backtracking:

Testing edit distance:





**Table:**

**Testing default graph**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **METHOD** | **Comparisons** | **Time** | **Comparisons** | **Time** | **Comparisons** | **Time** |
| **Randomization** | 20 | 0.004 | 24 | 0.004 | 73 | 0.013 |
| **Backtracking** | 8 | 0.0009 | 8 | 0.0009 | 8 | 0.0009 |

**Testing graph with no path**

|  |  |  |
| --- | --- | --- |
| **METHOD** | **Comparisons** | **Time** |
| **Randomization** | 1024 | 0.048 |
| **Backtracking** |  |  |

Backtracking was always constant because it reached the solution at the same time, it makes the same steps everytime. In theory randomization can be faster sometimes, but in average for the default graph it was not.

Backtracking would need to test all possible results to conclude that no path was found. Randomization would depend on the input.

**IV. Conclusions**

Depending on your problem to solve and what you would rather your solution algorithm to have, be it speed, or efficiency, you might want to approach a solution differently. Using these algorithms might mean sacrificing accuracy but if it is good enough a certain percentage of the time, then it is worth it to implement most of the times. I learnt that there are many ways to solve a problem and algorithms that can be adapted to a particular problem. I realized with this lab that there are other algorithms I’ve created that would have been more efficient implementing these methods.

**V. Appendix – Source code**

**--Lab 7—**

"""

COURSE: CS 2302 Data Structures

AUTHOR: Elisa Jimenez Todd

ASSIGNMENT: Lab 7 - Algorithms

INSTRUCTOR: Olac Fuentes

TA: Anindita Nath

DATE: 12/4/2019

Program: This program lets the user implement finding a hamiltonian cycle on a

graph with randomization and backtracking. It lets the user use the default graph,

create a new graph, o generate a random graph. It returns the edges connected

that make the hamiltonian cycle or none if none was found. It also shows the

comparisons made and the time taken to complete.

This program also implements the edit distance algorythm with the exception that

only replacements between vowels or between consonants are allowed.

This prints the edit distance, the size of the matrix, and the time.

"""

import numpy as np

import graph\_EL as graph

import connected\_components as cc

import time

#randomized hamiltonian

def RandomizedHamiltonian(V,E): #receives number of vertices and edges

comparisons = 0 #counts the comparisons of edges in a cycle

for i in range(2\*\*(len(E))): #number of tries

comparisons +=1

Eh = randomSet(V,E) #random subset of E of size V

graphT = makeGraph(V,Eh) #makes graph with the edges Eh

#if graph (V,Eh) has 1 connected component and the in-degree of every vertex in V is 2:

if cc.connected\_components(graphT) == 1 and checkDeg2(graphT):

return Eh, comparisons # Eh forms a Hamiltonian cycle

return None, comparisons # No Hamiltonian cycle was found

def randomSet(size, edges):

elem = set() #edges to form the cycle

if len(edges)<size:

return elem

while len(elem)<size:

j = np.random.randint(len(edges)) #random index

repeat = False

for i in elem: #check if the edge has been already added

if edges[j].source == i.dest and edges[j].dest == i.source:

repeat = True

break

if not repeat:

elem.add(edges[j]) #add edge

return elem

#Backtracking Hamiltonian

def BacktrackingHamiltonian(V,E):

bgraph = graph.Graph(V)

return AuxBacktrackingHamiltonian(bgraph, list(E))

def AuxBacktrackingHamiltonian(bgraph, E):

global bcomparisons

if len(bgraph.el) == bgraph.vertices\*2:

bcomparisons += 1

if cc.connected\_components(bgraph) == 1 and checkDeg2(bgraph):

return []#if goal has been reached

else: #dod not make a hamiltonian cycle

return None

if len(E) <= 0: # No path could be formed

return None

edge = E[0]

bgraph.insert\_edge(edge.source, edge.dest)

curEdges = AuxBacktrackingHamiltonian(bgraph,E[2:]) #Take first edge

if curEdges is not None:

return [edge] + curEdges

bgraph.delete\_edge(edge.source, edge.dest)

return AuxBacktrackingHamiltonian(bgraph,E[2:]) #Do not take first edge

def makeGraph(V,Eh):

ngraph = graph.Graph(V)

for i in Eh:

ngraph.insert\_edge(i.source, i.dest) #inserts edges

return ngraph

def checkDeg2(G):

for i in range(G.vertices):

if G.in\_degree(i) != 2: #check in degree 2

return False

return True

#edit distance

def edit\_distance(s1,s2):

vowels = {'a','e','i','o','u'} #set of vowels

d = np.zeros((len(s1)+1,len(s2)+1),dtype=int) #matrix size s1 x s2

d[0,:] = np.arange(len(s2)+1) #s2 fills first row with int

d[:,0] = np.arange(len(s1)+1) #s1 fills first column " "

for i in range(1,len(s1)+1): # traverse each row

for j in range(1,len(s2)+1): # traverse each column

if s1[i-1] == s2[j-1]: #if letter on s1 is same as letter on s2

d[i,j] =d[i-1,j-1] #fill data with top left number

else:

n = [d[i,j-1],d[i-1,j-1],d[i-1,j]] #three adjacent numbers

if min(n) == d[i-1,j-1]: #top left replace

if (not s1[i-1] in vowels and s2[j-1] in vowels) or (s1[i-1] in vowels and not s2[j-1] in vowels):

n = [d[i,j-1],d[i-1,j]]

d[i,j] = min(n)+1 #fill data with min number + 1

return d[-1,-1] #return matrix

def makeRandomGraph(G, E):

V = G.vertices

for i in range(E):

G.insert\_edge(np.random.randint(V),np.random.randint(V))

return G

exit = False

g = graph.Graph(5)

g.insert\_edge(0,1)

g.insert\_edge(1,2)

g.insert\_edge(1,4)

g.insert\_edge(2,3)

g.insert\_edge(3,4)

g.insert\_edge(2,4)

g.insert\_edge(4,0)

while(not exit):

print("================================================")

print("Choose an algorithm to test:\n\t1) Randomization Hamiltonian\n\t2) Backtracking Hamiltonian\n\t3) Dynamic Programming\n\t4) Exit")

option = input(':')

if option == "1":

select = input("\t1) Test Default Graph\n\t2) Input Graph to test\n\t3) Create Random Graph\n:")

if select == "1":

gt = g

elif select == "2":

gt = graph.Graph(int(input("Vertices on graph:")))

for i in range(int(input("Edges to add:"))):

gt.insert\_edge(int(input("source:")), int(input("destination:")))

elif select == "3":

gt = graph.Graph(int(input("Vertices on graph:")))

gt = makeRandomGraph(gt, int(input("Edges to add:")))

gt.display()

start = time.time()

hamiltonian, comparisons = RandomizedHamiltonian(gt.vertices, gt.el)

end = time.time()

print("Comparisons made:", comparisons)

print("Time to find:", end-start)

if hamiltonian != None:

print("HAMILTONIAN CYCLE:")

print('[',end = "")

for i in hamiltonian:

print("(", i.source, ",", i.dest, ")", end ="")

print(']')

else:

print("No path found.")

elif option == "2":

select = input("\t1) Test Default Graph\n\t2) Input Graph to test\n\t3) Create Random Graph\n:")

if select == "1":

gt = g

elif select == "2":

gt = graph.Graph(int(input("Vertices on graph:")))

for i in range(int(input("Edges to add:"))):

gt.insert\_edge(int(input("source:")), int(input("destination:")))

elif select == "3":

gt = graph.Graph(int(input("Vertices on graph:")))

gt = makeRandomGraph(gt, int(input("Edges to add:")))

gt.display()

bcomparisons = 0

start = time.time()

bhamiltonian = BacktrackingHamiltonian(gt.vertices, gt.el)

end = time.time()

print("Comparisons made:", bcomparisons)

print("Time to find:", end-start)

if bhamiltonian != None:

print("HAMILTONIAN CYCLE:")

print('[',end = "")

for i in bhamiltonian:

print("(", i.source, ",", i.dest, ")", end ="")

print(']')

else:

print("No path found.")

elif option == "3":

select = input("\t1) Test default set of words\n\t2) Input words to test\n:")

if select == "1":

print("'sand', 'sound' :", edit\_distance('sand', 'sound'))

print("'abcd', 'abcd' :", edit\_distance('abcd', 'abcd'))

print("'sand', 'sound' :", edit\_distance('races', 'race'))

print("'races', 'race' :", edit\_distance('aeiou', 'aaiiu'))

print("'aeiou', 'aaiiu' :", edit\_distance('aeiou', 'abicu'))

print("'aeiou', 'bcdfg' :", edit\_distance('aeiou', 'bcdfg'))

else:

s1 = input("first word: ")

s2 = input("second word: ")

start = time.time()

dist = edit\_distance(s1, s2)

end = time.time()

print("EDIT DISTANCE:",dist)

print("# of operations :", len(s1)\*len(s2))

print("Time: ", end - start)

elif option == "4":

exit = True

else:

print("Select a number from 1 to 4.")

**#EDGE LIST-----**

import numpy as np

import matplotlib.pyplot as plt

import math

from scipy.interpolate import interp1d

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 in\_degree(self, v):

deg = 0

for edge in self.el:

if edge.dest == v:

deg += 1

return deg

**--DFS—**

from scipy.interpolate import interp1d

import numpy as np

import matplotlib.pyplot as plt

class DSF:

# Constructor

def \_\_init\_\_(self, sets):

# Creates forest with 'sets' root nodes

self.parent = np.zeros(sets,dtype=int)-1

def find(self,i):

# Returns root of tree that i belongs to

if self.parent[i]<0:

return i

return self.find(self.parent[i])

def union(self,i,j):

# Makes root of j's tree point to root of i's tree if they are different

# Return 1 if a parent reference was changed, 0 otherwise

root\_i = self.find(i)

root\_j = self.find(j)

if root\_i != root\_j:

self.parent[root\_j] = root\_i

return 1

return 0

**--CONNECTED COMPONENTS—**

import graph\_EL as graph

import dsf

def connected\_components(g):

vertices = g.vertices

components = vertices

s = dsf.DSF(vertices)

for edge in g.el:

components -= s.union(edge.source,edge.dest)

return components

**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