**LAB 7**

**DANIEL CRUZ**

**Introduction**

This lab is meant to show possible algorithm types used to tackle hard to solve problems. In this case, randomization and backtracking are used to solve the NP-complete problem of finding a Hamiltonian cycle. Dynamic programming, another algorithm type, will also be used in this lab to find the edit distance of two strings.

**Proposed Solution Design and Implementation**

In this lab I used several programs and functions that have been provided in previous labs or class. These include the graph representations adjacency list and edge list and the connected components program using disjoint set forest.

To tackle the Hamiltonian cycle problem using randomization, I created a method that takes in a graph as an adjacency list and an optional parameter of how many trials to run the randomization with. I sought to use the pseudocode presented in the lab instructions. In order to make the graph easier to work with, I converted it into an edge list representation. Next, for however many trials specified the following occurs: A new graph gh will be created. Randomly chosen edges will be inserted into this new graph until there are as many edges as vertices in the graph. It will then check if it has one connected component and if each vertex has an in-degree of 2. If this is the case, it returns the new graph, else it keeps randomly checking for the Hamiltonian cycle.

Solving for the Hamiltonian cycle with backtracking takes a similar approach. The inputted graph is assumed to be of type adjacency list and is converted to an edge list. The graph along with an empty graph is then sent to a recursive function that does the following: an edge from the graph will be inserted into the empty graph. If a graph with 5 edges is built, it will check if connected components is 1 and if all vertices have in-degree of 2. If this is true then the recursion ends. Otherwise, the backtracking algorithm will try different combinations of edges until it exhausts all combinations or it finds a Hamiltonian cycle.

The edit distance function is very similar as the one described in class. However, two lists are made that contain the set of vowels and consonants respectively. A matrix is built with the length of the two strings, however for every character it checks, it first checks if both characters are equal and only brings in the upper diagonal if true. Otherwise it checks if both characters are consonants or vowels and then finds the minimum of the left, top, or diagonal upper left and adds one to that value. Otherwise it only allows the minimum of the left or top and adds one.

**Experimental Results**

For both Hamilton cycle functions, the following graph will be inputted:

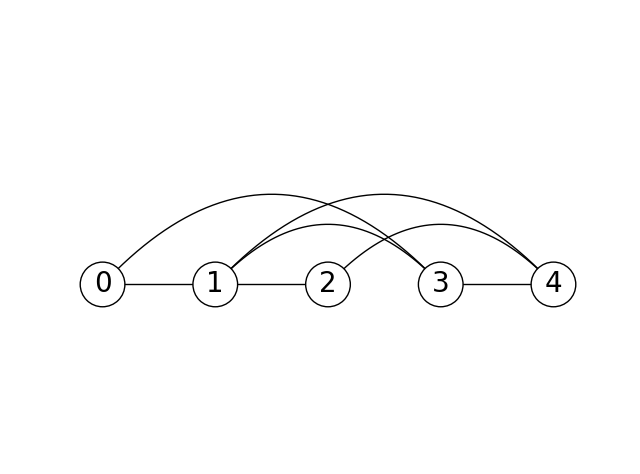
(0)--(1)--(2)

| / \ |

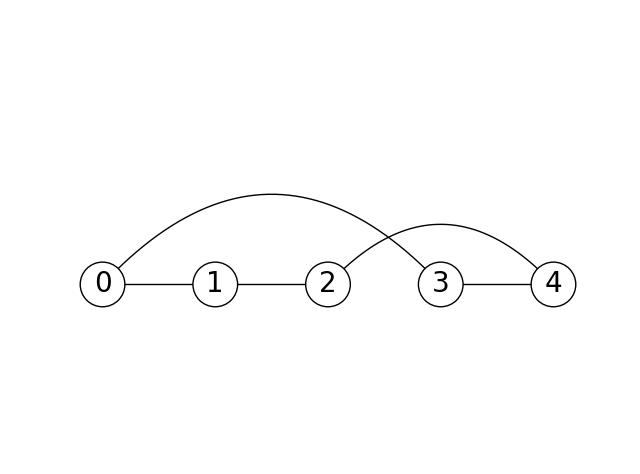
| / \ |

| / \ |

(3)-------(4)

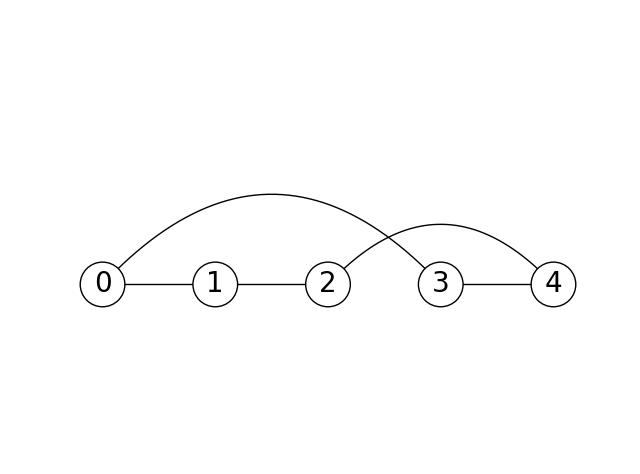
This value has a confirmed Hamilton cycle of {0,1,2,3,4}. 

Using the random algorithm, here is the found Hamilton cycle:



This matches with the expected output of the Hamilton cycle. There is only one connected component and each vertex has in-degree 2.

Similarly, here is the output with the backtracking version:



Now to test the same two functions for a graph that does not have a Hamiltonian cycle, I changed the graph as such:

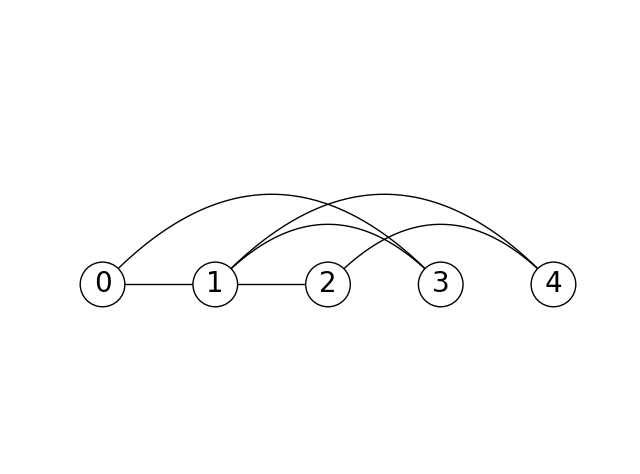
(0)--(1)--(2)

| / \ |

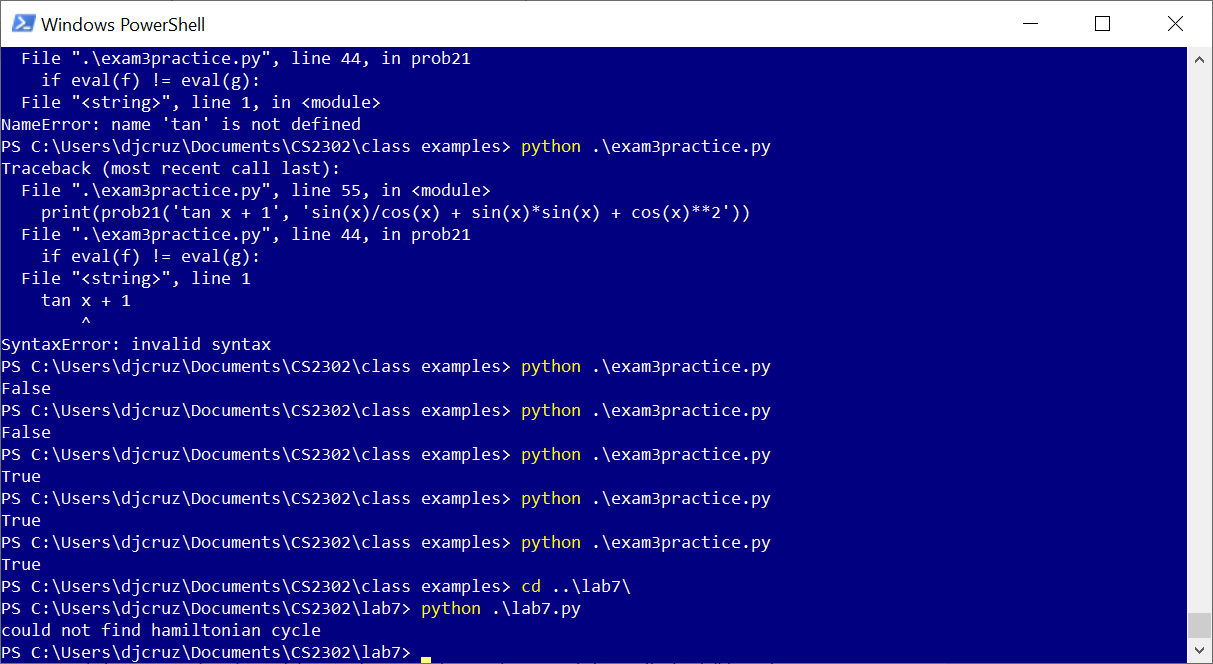
| / \ |

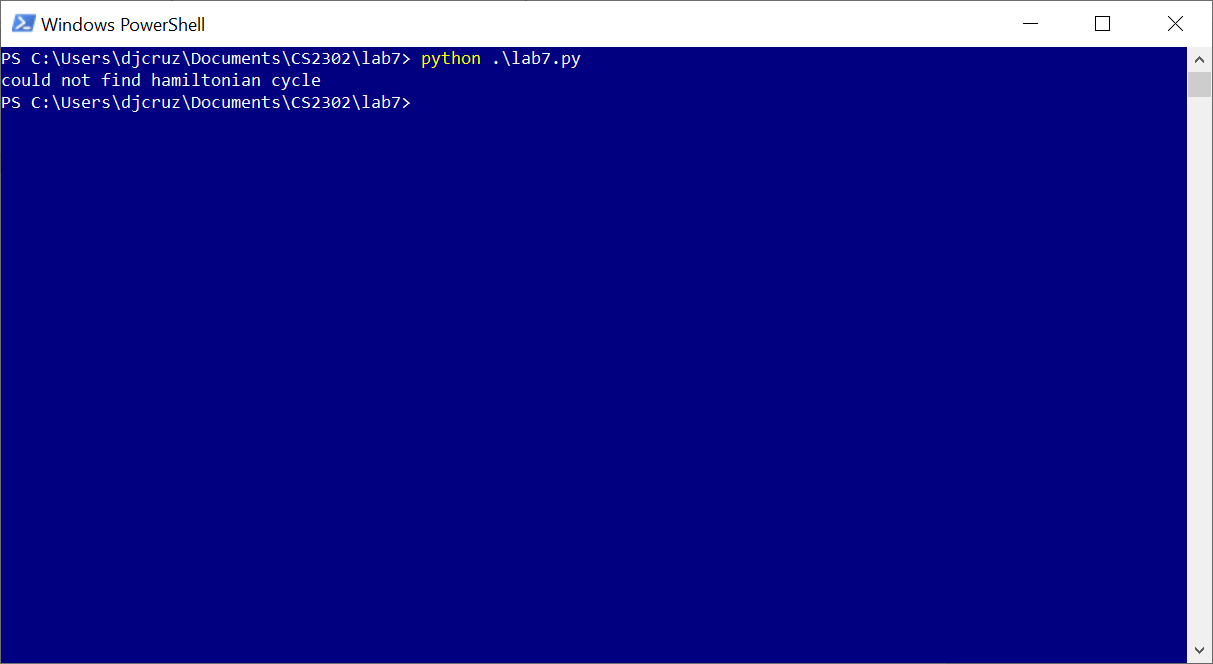
| / \ |

(3) (4)

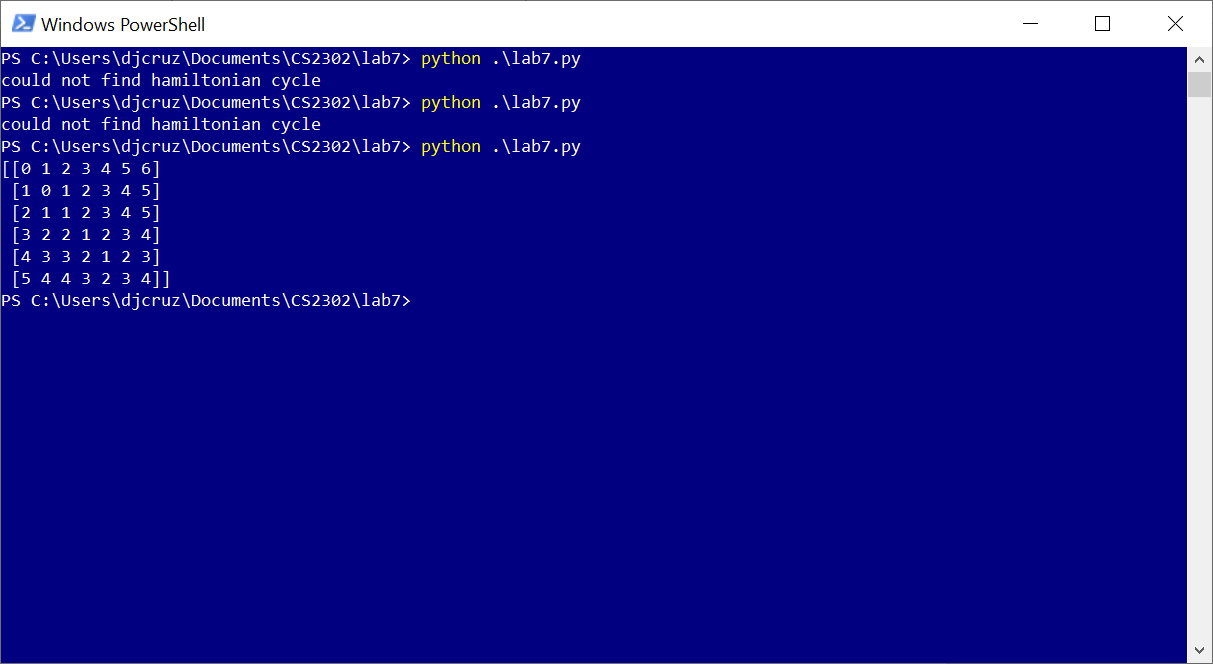


Here the cycle is broken and there is no Hamilton cycle. Here are the outputs of the two functions now:

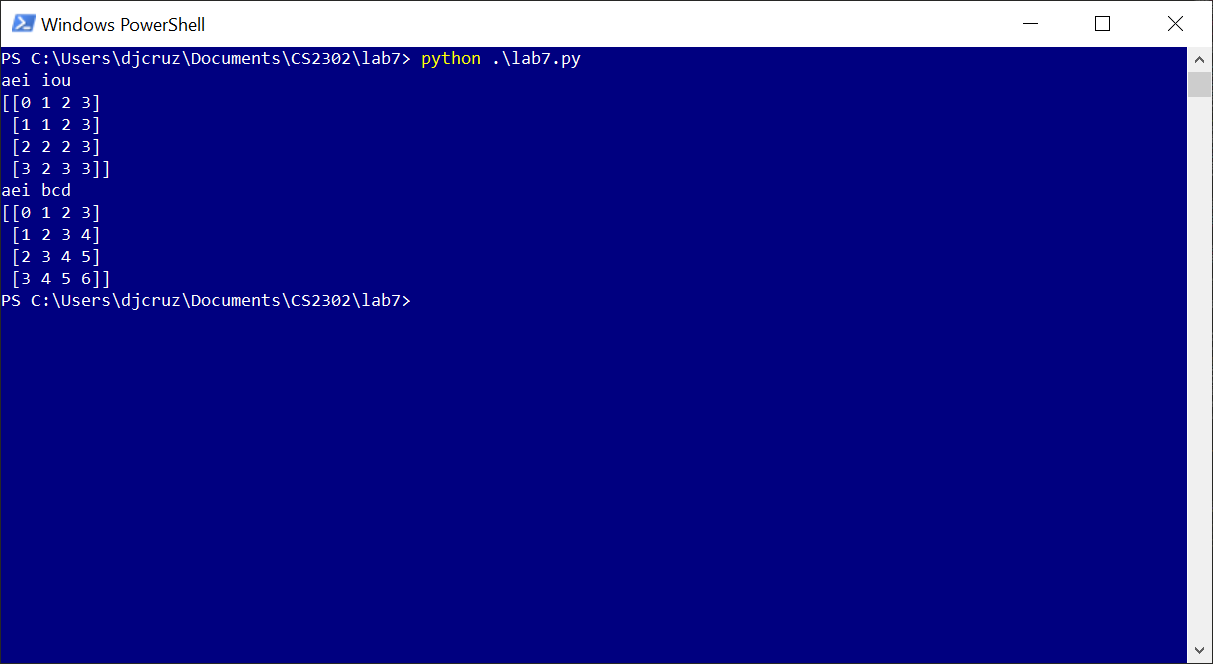




As for the dynamic programming, I tested the output with the two strings “Miners”, “Money”. When calculated by hand, there should be four edits as opposed to three edits from replacing ‘r’ with ‘y.’



The output correctly shows the appropriate actions based on the letters and returns four as expected. To test that no replacements are valid between vowels and consonants, “aei” with “iou” and “aei” with “bcd” will be tested. The should both return different numbers, with the first set taking 3 edits and the second taking 6.



**Conclusion**

Three algorithms we learned in class are backtracking, randomization, and dynamic programming. Backtracking and randomization in particular can be used to attempt to solve NP-complete problems. While using these two algorithms to solve the NP-complete problem of Hamiltonian cycles, I found backtracking much more thorough however much slower in larger graphs. Randomization is much more reliable in terms of time yet it has the possibility to return false if it cannot find a cycle in the allotted tries. Dynamic programming involves using previous calculation to further the current calculation. When applied to an edit distance algorithm, the number of changes to change one string to another can be calculated.

import graph\_AL as graph

import numpy as np

from connected\_components import connected\_components

#finds in-degrees of vertex v

def in\_degree(G, v):

sumDeg = 0

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

for e in G.al[i]:

if e.dest == v:

sumDeg += 1

return sumDeg

#hamiltonian cycle using randomization

def randomizedHamiltonian(G, trials = 1000):

gEL = G.as\_EL()

for q in range(trials):

gh = graph.Graph(len(G.al), weighted=G.weighted, directed=G.directed)

gh = gh.as\_EL()

edges = gEL.el[:]

#randomly choose edges

for r in range(gh.vertices):

gh.el.append(edges.pop(np.random.randint(len(edges))))

#convert to adjacency list to check connected components

gh = gh.as\_AL()

components,s = connected\_components(gh)

if components == 1:

has2Deg = True

#check if each vertex has in\_degree 2.

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

indeg = in\_degree(gh,i)

if indeg != 2:

has2Deg = False #give up, not a hamiltonian cycle

if has2Deg:

return gh

#could not find hamiltonian cycle.

return None

#takes in list and edge list.

def backtracker(g,gh):

#end recursion if out of edges

if len(gh.el) == gh.vertices:

ghAL = gh.as\_AL()

components,s = connected\_components(ghAL)

if components == 1:

#check in-degrees

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

if in\_degree(ghAL, i) != 2:

return None

return ghAL

return None

if len(g) == 0:

return None

else:

nextEdge = g[0]

gh.el = gh.el + [nextEdge]

response = backtracker(g[1:], gh)

if response is not None:

return response

gh.el.remove(nextEdge) # get rid of inserted edge and backtrack

response = backtracker(g[1:], gh)

return response

def backtrackHamiltonian(G):

# #check if every vertex in V has in-degree 2.

gEL = G.as\_EL()

for e in gEL.el:

print(e.source, e.dest, e.weight)

gBuild = graph.Graph(len(G.al), weighted=G.weighted, directed=G.directed)

return backtracker(gEL.el,gBuild.as\_EL())

def editDistance(word1, word2):

vowels = ['a','e','i','o','u']

consonants = ['b','c','d','f','g','h','j','k','l','m','n','p','q','r','s','t','v','w','x','z']

matrix = np.zeros((len(word1)+1,len(word2)+1),dtype=int)

for i in range(len(matrix)):

matrix[i][0] = i

for i in range(len(matrix[0])):

matrix[0][i] = i

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

for j in range(1,len(matrix[i])):

#if characters are the same

if word1[i-1] == word2[j-1]:

matrix[i][j] = matrix[i-1][j-1]

else:

#if both characters are vowels or consonants, find the minimum and add 1

if (word1[i-1] in vowels and word2[j-1] in vowels) or (word1[i-1] in consonants and word2[j-1] in consonants):

matrix[i][j] = min(matrix[i-1][j], matrix[i][j-1], matrix[i-1][j-1])+1

else:

#if both characters not vowel or consonant, then replacements (upper left) are not possible.

matrix[i][j] = min(matrix[i-1][j], matrix[i][j-1])+1

print(matrix)

def testHamiltonianCycle():

# g = graph.Graph(3, weighted=False, directed=False)

# g.insert\_edge(0,1)

# g.insert\_edge(1,2)

# g.insert\_edge(2,0)

g = graph.Graph(5, weighted=False, directed=False)

g.insert\_edge(0,1)

g.insert\_edge(1,2)

g.insert\_edge(2,4)

g.insert\_edge(1,4)

g.insert\_edge(1,3)

g.insert\_edge(0,3)

g.insert\_edge(3,4) #comment this edge to get rid of hamiltonian cycle

g.draw()

#hamilCyc = randomizedHamiltonian(g)

hamilCyc = backtrackHamiltonian(g)

if hamilCyc is not None:

print('Found hamiltonian cycle')

hamilCyc.draw()

else:

print("could not find hamiltonian cycle")

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

testHamiltonianCycle()

#editDistance("money", "miners")

#print("aei", "iou")

#editDistance("aei", "iou")

#print("aei","bcd")

#editDistance("aei", "bcd")

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