LAB REPORT #7

Sergio Ortiz
December 4, 2019

Professor – Olac Fuentes
TA – Anindita Nath
CS 2302 Data Structures – Fall 2019

Introduction

In this lab, we were asked to showcase several different algorithm design techniques by implementing two different algorithms. First, we were asked to find a Hamiltonian Cycle in a graph using randomization, and then, using backtracking. After this, we were asked to modify a dynamic programming algorithm consisting of finding the distance between two words, only considering two characters different if they were both vowels or both consonants.

Proposed Solution Design and Implementation

Randomized Algorithms:

My idea to solve the Hamiltonian Cycle problem using randomization was to build random subsets of the set of edges and then check if these were a Hamiltonian Cycle. I would do this a specific number of times, and based on my experimentation, 1000 seemed to do the trick, although we could easily increase or decrease this. In order to randomly build a subset, I would basically visit every vertex and randomly visit one of its vertices. I would do this in a loop that lasted as long as the length of the graph, in other words, how many vertices the graph had. This was because a Hamiltonian Cycle only needs to go through each vertex once, so this size of loop was enough to fit the Hamiltonian Cycle if one existed. Then, once I had a randomly generated subset, I would check if it was a Hamiltonian Cycle by checking if the in – degree of every vertex was 2 and if the graph had one connected component. If both of these conditions were satisfied, a Hamiltonian Cycle had been found.

Backtracking:

This was perhaps the hardest algorithm for me to implement. My general idea was to traverse the graph, visiting each vertex's adjacent vertices and going back if this adjacent vertex did not meet the Hamiltonian Cycle requirements. In this way, we would be building the Hamiltonian Cycle as we went and would go back and try with other vertices if the current path did not satisfy the conditions. It is basically like a brute force solution, which is essentially what backtracking is. If we reached the end of the graph and the conditions for a Hamiltonian Cycle were met, then the algorithm had successfully found a Hamiltonian Cycle. Otherwise, the algorithm could guarantee that there was no cycle in the graph, since it had already traversed all possible paths. This was the worst case runtime, since it would go through the entire graph only to return False. I could not come up with a way to minimize this worst case runtime but I am pretty sure it was expected for us to do so.

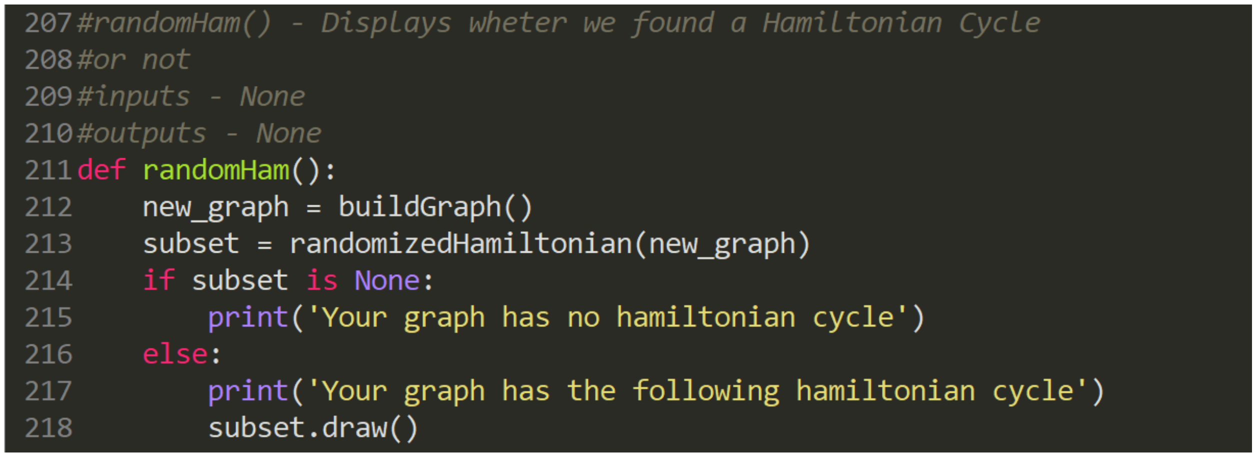
Dynamic Programming:

My general idea for this algorithm was quite simple and I figured it would be a breeze, but some slight complications, as always, came up. The instructions only asked us to modify the already provided edit distance function to only swap characters if they were either both vowels or both consonants. Therefore, I created a list with all the vowels and my idea was to simply put a set of if statements that would check if both characters were either both in this list or both not in the list before adding one to the distance. If neither of these conditions were met, then we would simply treat the letters as if they were the same. This worked fine for words that were of same length, but I found out that the algorithm did this strange thing were it would recycle old letters if one of the words was shorter. This was problematic because, if one of the words was shorter, then it automatically had the difference in length as distance between the two. However, my algorithm would not compute this because it was still checking if the recycled letter and the current letter were both vowels or consonants before adding to the difference. Therefore, I solved this by using a hash table because I love hash tables in python. I created two hash tables, which each had the characters of each specific word as keys and the number of times they appeared in the word was values. Each time we visited a specific letter, I would subtract one from its respective hash table's key. Then, before visiting a specific letter, I would check if its key value was zero. If so, I would simply proceed with the regular algorithm because this meant one word was shorter so there was no need to check if they were both vowels and consonants.

Experimental Results

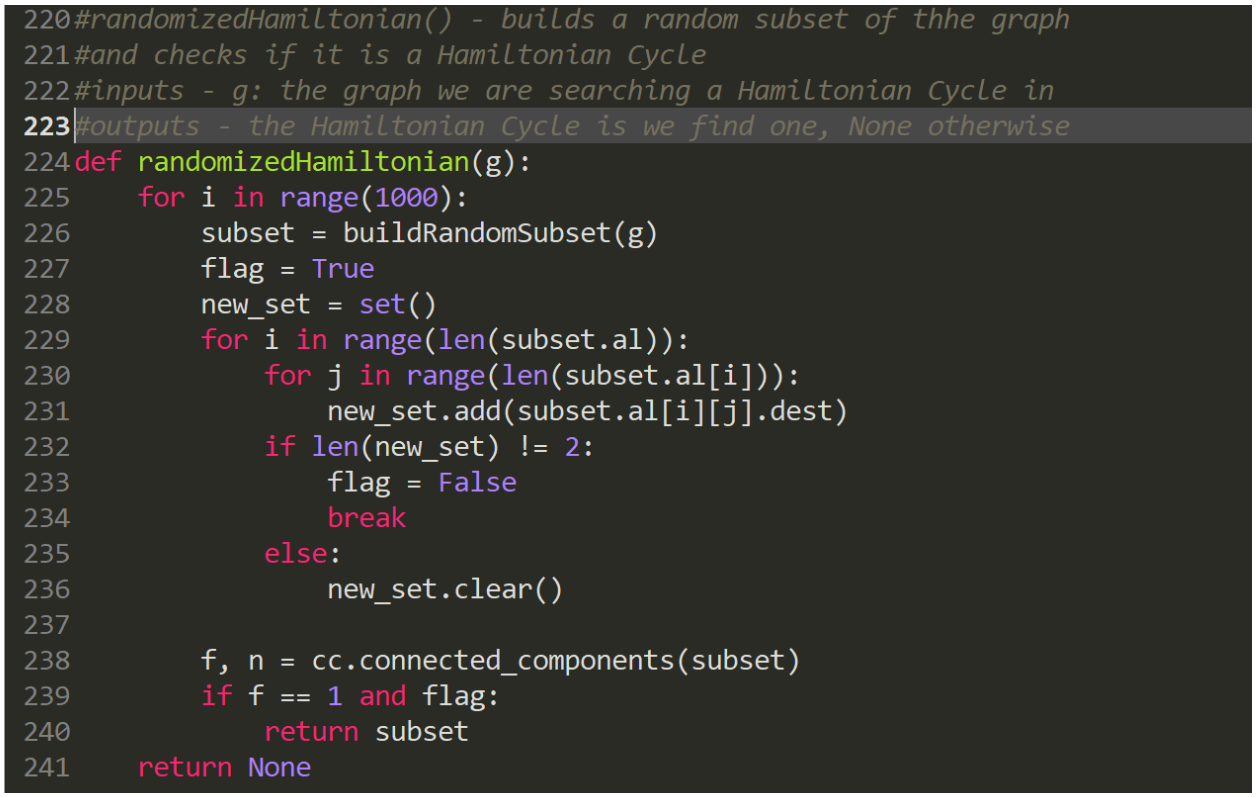
Randomized Algorithms:

The first method I created to solve this mainly covered the aesthetics of the algorithm. It called another method which would return a subset if one existed and print this out.



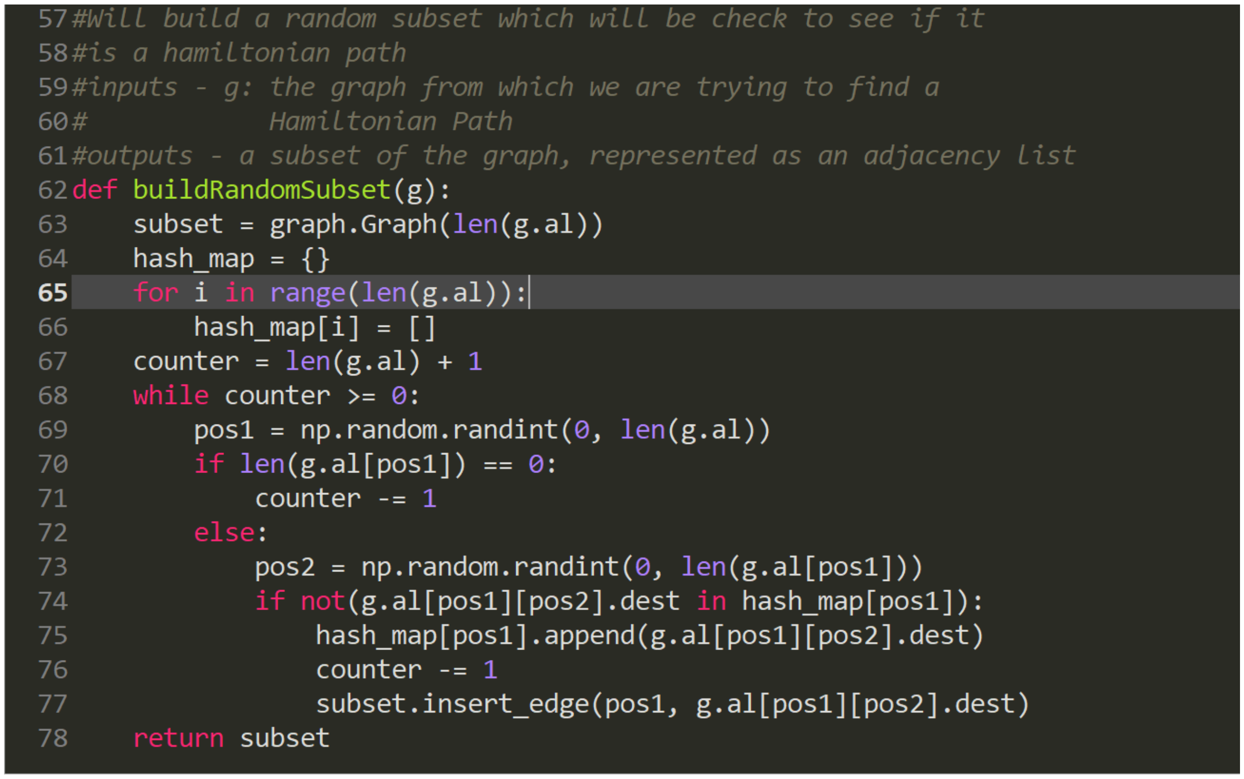
```python
#randomHam() - Displays wheter we found a Hamiltonian Cycle
#or not
#inputs - None
#outputs - None
def randomHam():
    new_graph = buildGraph()
    subset = randomizedHamiltonian(new_graph)
    if subset is None:
        print('Your graph has no hamiltonian cycle')
    else:
        print('Your graph has the following hamiltonian cycle')
        subset.draw()
```

The method called to assign a value to subset is where the main algorithm occurs. In a nutshell, it creates 1000 random subsets and checks if they are a Hamiltonian Cycle. As soon as it encounters one that is a Hamiltonian Cycle, it exits the loop and returns this graph. The way it checks if it is a valid is by seeing if the in degree of every vertex is 2 and also if it contains exactly one connected component.



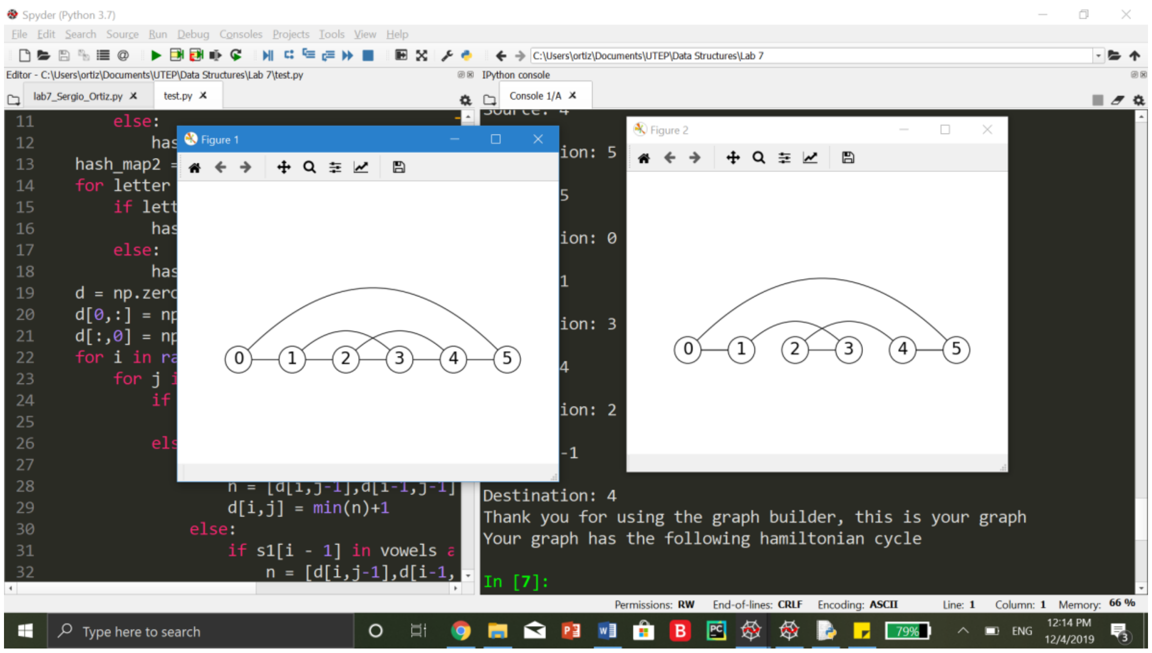
```python
#randomizedHamiltonian() - builds a random subset of thhe graph
#and checks if it is a Hamiltonian Cycle
#inputs - g: the graph we are searching a Hamiltonian Cycle in
#outputs - the Hamiltonian Cycle is we find one, None otherwise
def randomizedHamiltonian(g):
    for i in range(1000):
        subset = buildRandomSubset(g)
        flag = True
        new_set = set()
        for i in range(len(subset.al)):
            for j in range(len(subset.al[i])):
                new_set.add(subset.al[i][j].dest)
            if len(new_set) != 2:
                flag = False
                break
            else:
                new_set.clear()

        f, n = cc.connected_components(subset)
        if f == 1 and flag:
            return subset
    return None
```

The method that builds a random subset first creates a new graph of length the number of vertices in the original graph. Then, I created a hash map because I love them and made the keys the vertices and the value a list containing all the adjacent vertices.



```python
#Will build a random subset which will be check to see if it
#is a hamiltonian path
#inputs - g: the graph from which we are trying to find a
#            Hamiltonian Path
#outputs - a subset of the graph, represented as an adjacency list
def buildRandomSubset(g):
    subset = graph.Graph(len(g.al))
    hash_map = {}
    for i in range(len(g.al)):
        hash_map[i] = []
    counter = len(g.al) + 1
    while counter >= 0:
        pos1 = np.random.randint(0, len(g.al))
        if len(g.al[pos1]) == 0:
            counter -= 1
        else:
            pos2 = np.random.randint(0, len(g.al[pos1]))
            if not(g.al[pos1][pos2].dest in hash_map[pos1]):
                hash_map[pos1].append(g.al[pos1][pos2].dest)
                counter -= 1
                subset.insert_edge(pos1, g.al[pos1][pos2].dest)
    return subset
```

I would later use this to check if the current vertex was adjacent to the previous one. I would keep doing this until we filled the graph. Then, this graph would be checked to see if it was a Hamiltonian Cycle. Here is a sample run of the program.


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\ortiz\Documents\UTEP\Data Structures\Lab 7
Editor - C:\Users\ortiz\Documents\UTEP\Data Structures\Lab 7\test.py
lab7_Sergio_Ortiz.py
test.py
IPython console
Console 1/A
Figure 1
Figure 2
0
1
2
3
4
5
Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 1 Column: 1 Memory: 66 %
Type here to search
12:14 PM
12/4/2019

```
        else:
            has
    hash_map2 =
    for letter
        if lett
            has
        else:
            has
    d = np.zerc
    d[0,:] = np
    d[:,0] = np
    for i in ra
        for j i
            if

            els

                n = [d[i,j-1],d[i-1,j-1]
                d[i,j] = min(n)+1
            else:
                if s1[i - 1] in vowels a
                    n = [d[i,j-1],d[i-1,
```

```
Source: 4
ion: 5
5
ion: 0
1
ion: 3
4
ion: 2
-1
Destination: 4
Thank you for using the graph builder, this is your graph
Your graph has the following hamiltonian cycle

In [7]:
```

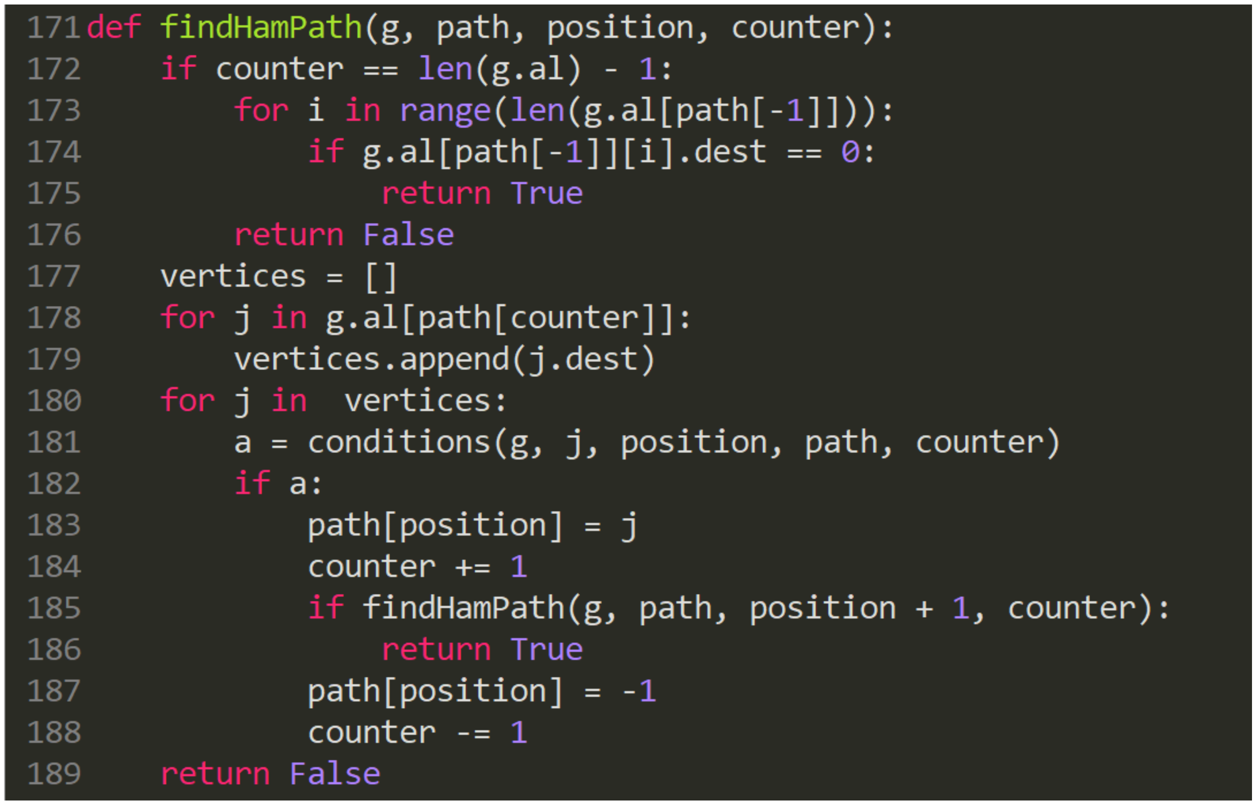
The graph on the left is the original graph while the one on the right is the found Hamiltonian Cycle. Of course, in this specific case, this is not the only one that exists in the graph, but it is the first one that the randomized nature of the algorithm found.

Backtracking:

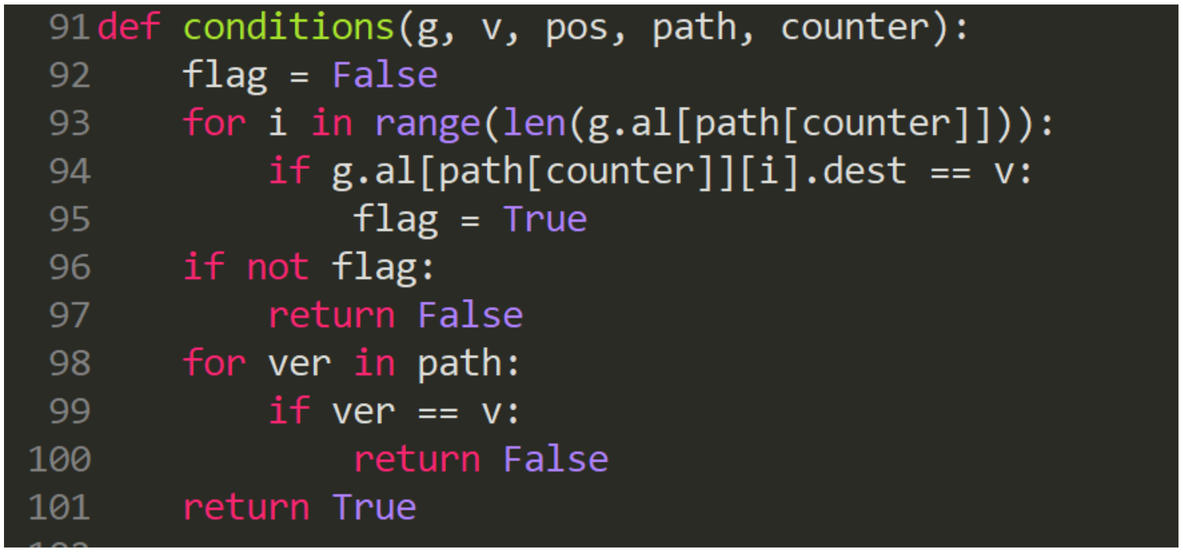
The first part of my implementation, again, took care mainly of the aesthetics of the algorithm. It would first ask the user to build a graph and then call the backtracking Hamiltonian algorithm to check if it had a Hamiltonian Cycle. If so, it would build a new graph to show the user based off the path list that the method would return.

```
#backtrackingHamiltonian() - Takes care of all the aesthetics
#for the hamiltonian path using backtracking. Initializes all the tools
#the algorithm will use and then prints our the result and shows the
#hamiltonian path if one exists
#inputs - None
#outputs - None
def backtrackingHamiltonian():
    new_graph = buildGraph()
    counter = 0
    path = [-1] * len(new_graph.al)
    path[0] = 0
    if findHamPath(new_graph, path, 1, counter) == False:
        print('Sorry, no Hamiltonian Path')
    else:
        print('We found the following Hamiltonian Path')
        new_graph = graph.Graph(len(path))
        for i in range(len(path) - 1):
            new_graph.insert_edge(path[i], path[i + 1])
        new_graph.insert_edge(path[-1], path[0])
        new_graph.draw()
```

The first thing that the method to find a Hamiltonian Cycle does is to check if we are on the last vertex of the path. If so, this vertex has to be connected to the first one in the path, so we check for this. If this condition is met, the method returns True. It returns False otherwise. Then, we check all the vertices that the current vertex is adjacent with and choose each of them sequentially. For each adjacent vertex, we send it to another method called conditions, which checks if adding the vertex will not violate the properties of a Hamiltonian Cycle. It checks if the first and last vertex in the path are adjacent if we are on the last vertex. It also checks if the vertex is not already in the path, which would violate the properties of a Hamiltonian Cycle.

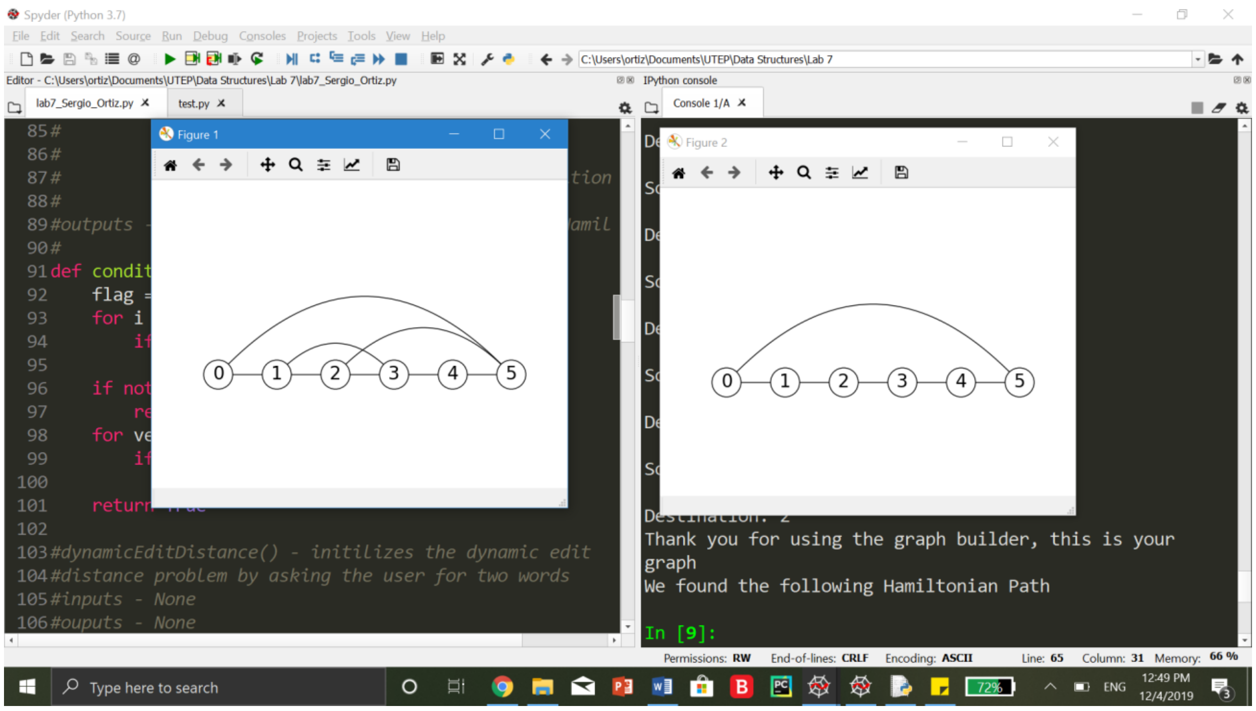


```
def findHamPath(g, path, position, counter):
    if counter == len(g.al) - 1:
        for i in range(len(g.al[path[-1]])):
            if g.al[path[-1]][i].dest == 0:
                return True
        return False
    vertices = []
    for j in g.al[path[counter]]:
        vertices.append(j.dest)
    for j in  vertices:
        a = conditions(g, j, position, path, counter)
        if a:
            path[position] = j
            counter += 1
            if findHamPath(g, path, position + 1, counter):
                return True
            path[position] = -1
            counter -= 1
    return False
```



```
def conditions(g, v, pos, path, counter):
    flag = False
    for i in range(len(g.al[path[counter]])):
        if g.al[path[counter]][i].dest == v:
            flag = True
    if not flag:
        return False
    for ver in path:
        if ver == v:
            return False
    return True
```

If the next vertex does not violate the properties of a Hamiltonian Cycle, we proceed along this path until we meet either then end of the cycle or a vertex that violates the condition. This is what lines 185 and 186 do. As soon as we come across an invalid vertex, we go back and try with the other adjacent vertices. This is what we do in lines 187 and 188, placing a negative on in the current position since we did not place a successful vertex here and also updating our position on the path list. If this method returns True, then we found a Hamiltonian Cycle. Here is a test run of the algorithm.



Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\ortiz\Documents\UTEP\Data Structures\Lab 7
Editor - C:\Users\ortiz\Documents\UTEP\Data Structures\Lab 7\lab7_Sergio_Ortiz.py
lab7_Sergio_Ortiz.py
test.py
IPython console
Console 1/A
Figure 1
Figure 2
0
1
2
3
4
5
Type here to search

```
#
#
#
#
#outputs -
#
def condit
    flag =
    for i
        if

    if not
        re
    for ve
        if

    return True

#dynamicEditDistance() - initilizes the dynamic edit
#distance problem by asking the user for two words
#inputs - None
#ouputs - None
```

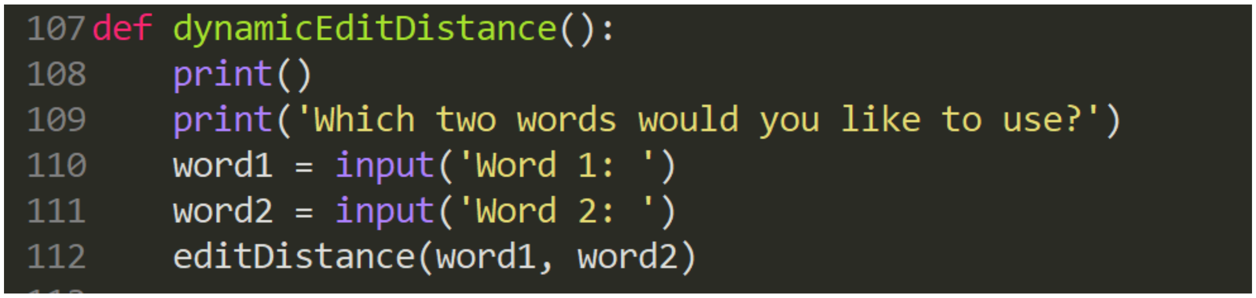
```
Destination: 2
Thank you for using the graph builder, this is your
graph
We found the following Hamiltonian Path

In [9]:
```

Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 65 Column: 31 Memory: 66 %
12:49 PM
12/4/2019

The graph on the left is the original and the one on the left is the found Hamiltonian Cycle.

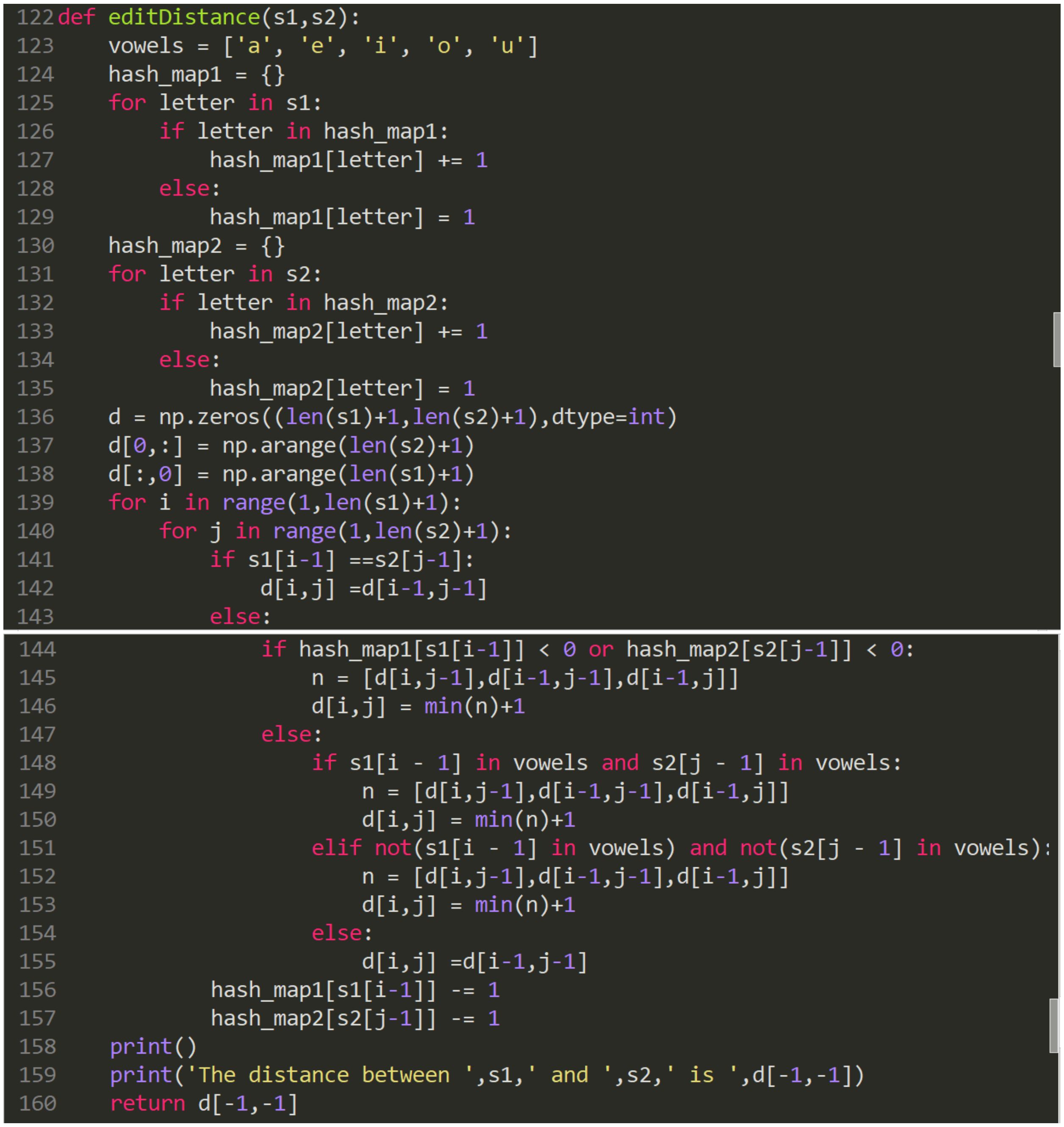
Dynamic Programming:

The first part of the algorithm simply asks the user what two words he wants to use.



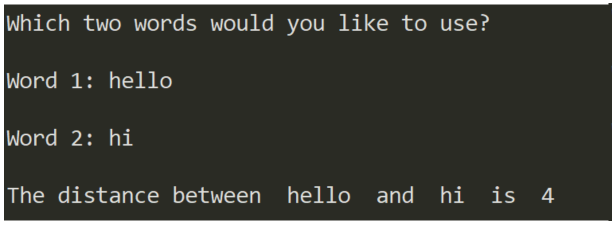
```
def dynamicEditDistance():
    print()
    print('Which two words would you like to use?')
    word1 = input('Word 1: ')
    word2 = input('Word 2: ')
    editDistance(word1, word2)
```

The edit distance function is where we use dynamic programming to check the distance between the two words. The first thing I did was to create two hash maps, each having the keys as the characters of the word and the values of each key being how often the character appeared in the word. This would be used to keep track if we had checked all the characters of a word. From here, the algorithm is the same as the one provided in class except that it checks if both of the letters are both vowels or consonants before adding to the distance. If not, then the algorithm will proceed as if the two letters were the same. Once it does this, it removes one from the number of occurrences of the current letters. If we come to a point where we are checking for a letter whose value is zero, this means that we already checked all those letters in the word, so the process of checking for vowel or consonant does not apply since these are simply extra letters. Therefore, if this was the case, I wrote the code so that the original algorithm would execute. In this way, we would not come across trouble if the two words were of differing lengths.

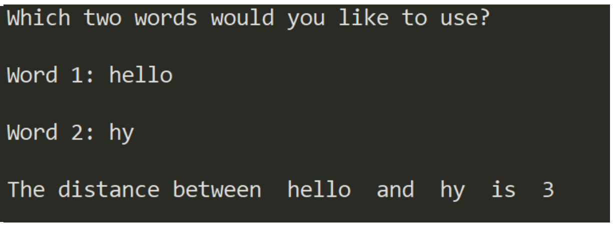


```
def editDistance(s1,s2):
    vowels = ['a', 'e', 'i', 'o', 'u']
    hash_map1 = {}
    for letter in s1:
        if letter in hash_map1:
            hash_map1[letter] += 1
        else:
            hash_map1[letter] = 1
    hash_map2 = {}
    for letter in s2:
        if letter in hash_map2:
            hash_map2[letter] += 1
        else:
            hash_map2[letter] = 1
    d = np.zeros((len(s1)+1,len(s2)+1),dtype=int)
    d[0,:] = np.arange(len(s2)+1)
    d[:,0] = np.arange(len(s1)+1)
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            if s1[i-1] ==s2[j-1]:
                d[i,j] =d[i-1,j-1]
            else:
                if hash_map1[s1[i-1]] < 0 or hash_map2[s2[j-1]] < 0:
                    n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
                else:
                    if s1[i - 1] in vowels and s2[j - 1] in vowels:
                        n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                        d[i,j] = min(n)+1
                    elif not(s1[i - 1] in vowels) and not(s2[j - 1] in vowels):
                        n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                        d[i,j] = min(n)+1
                    else:
                        d[i,j] =d[i-1,j-1]
            hash_map1[s1[i-1]] -= 1
            hash_map2[s2[j-1]] -= 1
    print()
    print('The distance between ',s1,' and ',s2,' is ',d[-1,-1])
    return d[-1,-1]
```

Here are some outputs of the algorithm. As we can see, it replaces the ‘e’ and ‘I’ but not the ‘e’ and ‘y’.



```
Which two words would you like to use?

Word 1: hello

Word 2: hi

The distance between  hello  and  hi  is  4
```

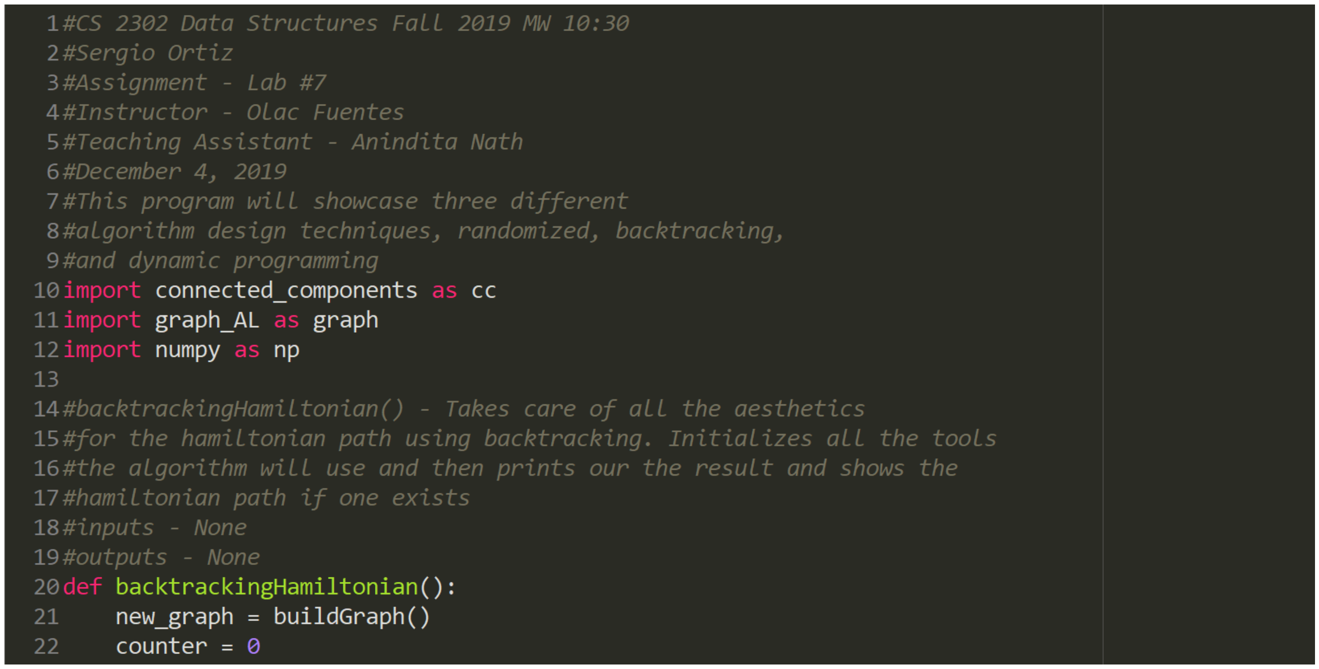


```
Which two words would you like to use?

Word 1: hello

Word 2: hy

The distance between  hello  and  hy  is  3
```

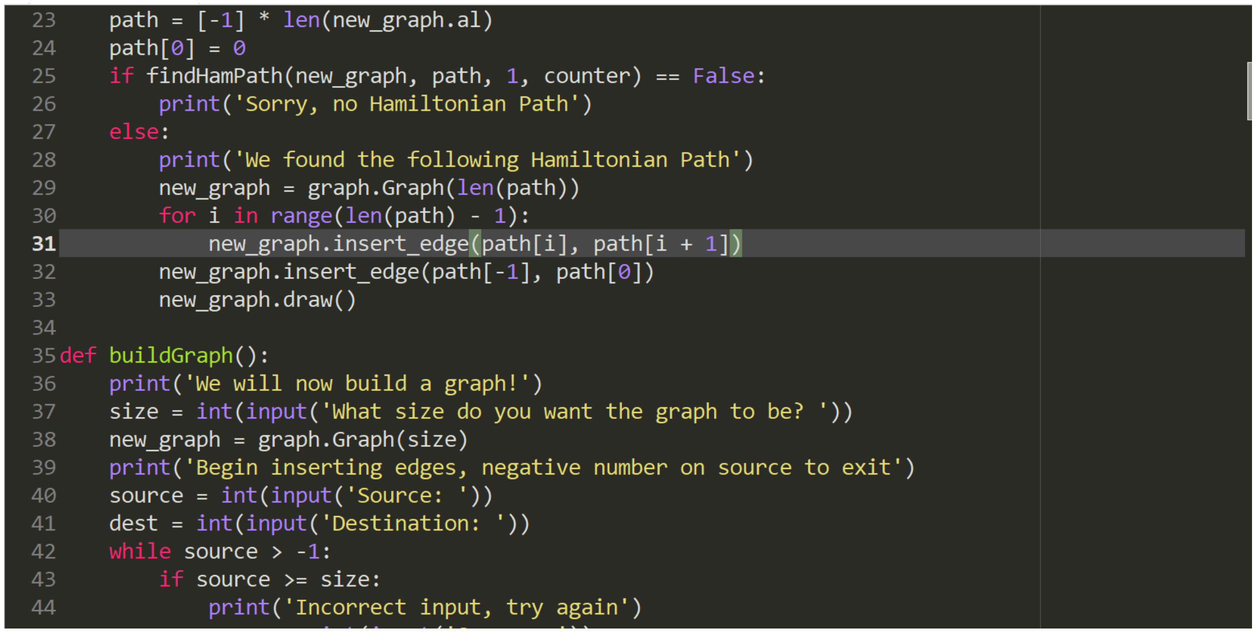
Conclusion

This lab was entertaining because it forced us to think like a computer scientist when forming these algorithms. The overall skeleton of the algorithms and their ideas were already formed for us, it was just a matter of seeing how to implement them. It was fun because these scenarios are similar to what a computer scientist does on a daily basis, solve problems by building algorithms. I feel like this lab really helped expand my problem solving toolbox because I now have different design techniques that could result useful in the future.

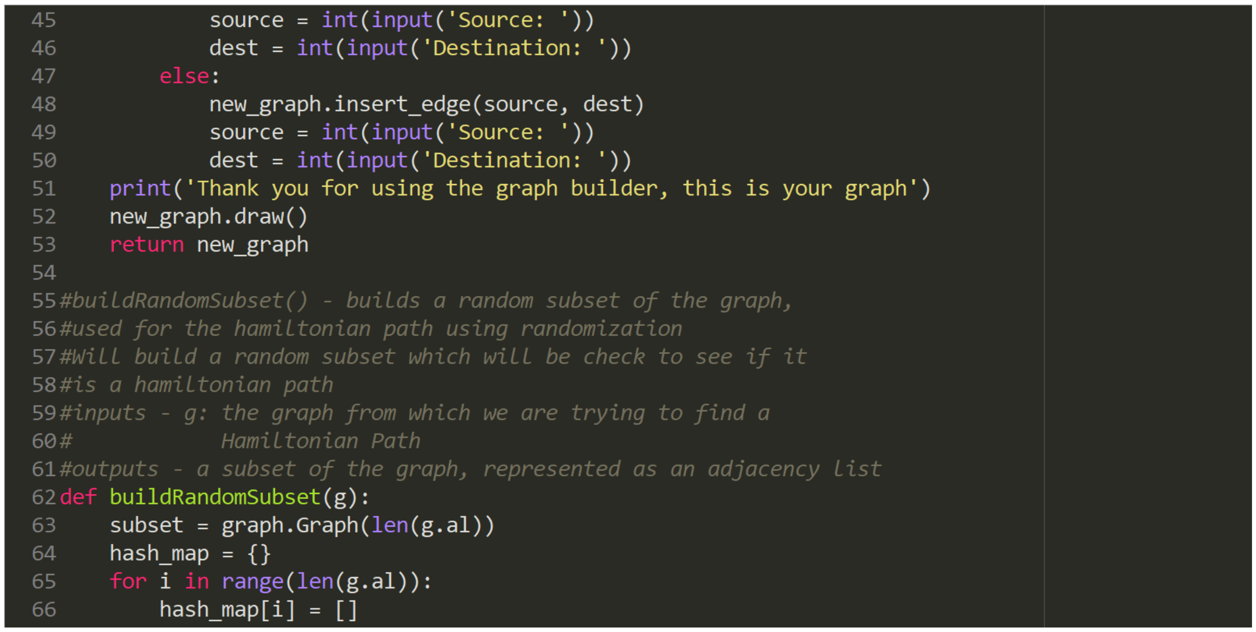
Appendix



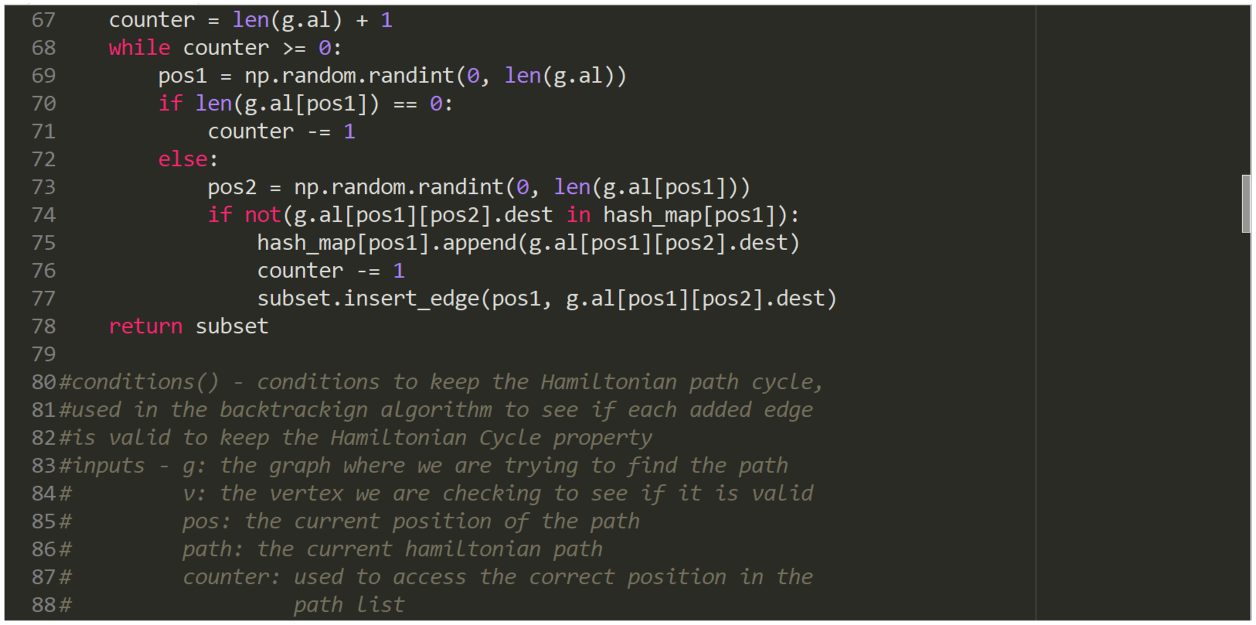
```python
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #7
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#December 4, 2019
#This program will showcase three different
#algorithm design techniques, randomized, backtracking,
#and dynamic programming
import connected_components as cc
import graph_AL as graph
import numpy as np

#backtrackingHamiltonian() - Takes care of all the aesthetics
#for the hamiltonian path using backtracking. Initializes all the tools
#the algorithm will use and then prints our the result and shows the
#hamiltonian path if one exists
#inputs - None
#outputs - None
def backtrackingHamiltonian():
    new_graph = buildGraph()
    counter = 0
```



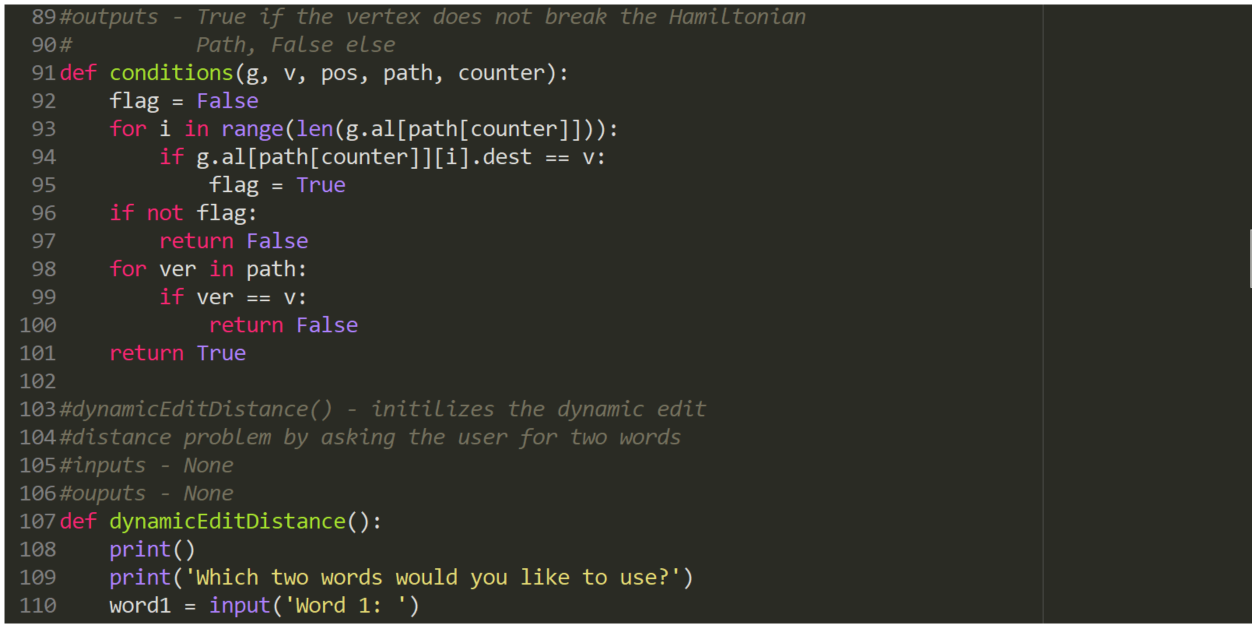
```python
    path = [-1] * len(new_graph.al)
    path[0] = 0
    if findHamPath(new_graph, path, 1, counter) == False:
        print('Sorry, no Hamiltonian Path')
    else:
        print('We found the following Hamiltonian Path')
        new_graph = graph.Graph(len(path))
        for i in range(len(path) - 1):
            new_graph.insert_edge(path[i], path[i + 1])
        new_graph.insert_edge(path[-1], path[0])
        new_graph.draw()

def buildGraph():
    print('We will now build a graph!')
    size = int(input('What size do you want the graph to be? '))
    new_graph = graph.Graph(size)
    print('Begin inserting edges, negative number on source to exit')
    source = int(input('Source: '))
    dest = int(input('Destination: '))
    while source > -1:
        if source >= size:
            print('Incorrect input, try again')
```



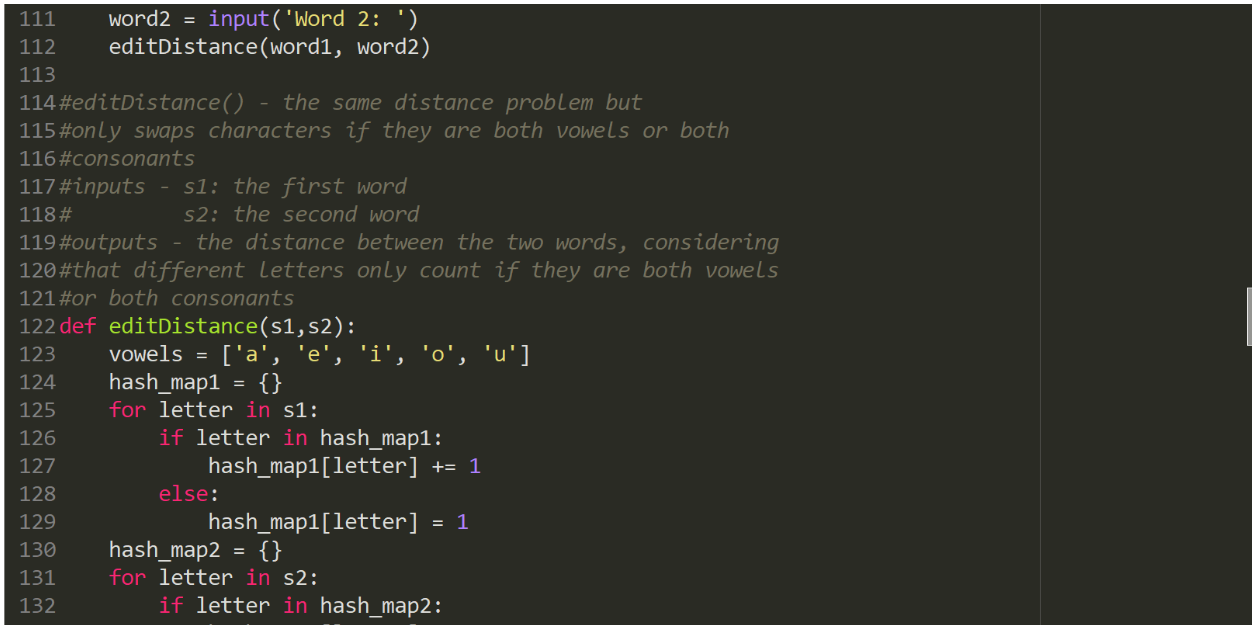
```python
            source = int(input('Source: '))
            dest = int(input('Destination: '))
        else:
            new_graph.insert_edge(source, dest)
            source = int(input('Source: '))
            dest = int(input('Destination: '))
    print('Thank you for using the graph builder, this is your graph')
    new_graph.draw()
    return new_graph

#buildRandomSubset() - builds a random subset of the graph,
#used for the hamiltonian path using randomization
#Will build a random subset which will be check to see if it
#is a hamiltonian path
#inputs - g: the graph from which we are trying to find a
#            Hamiltonian Path
#outputs - a subset of the graph, represented as an adjacency list
def buildRandomSubset(g):
    subset = graph.Graph(len(g.al))
    hash_map = {}
    for i in range(len(g.al)):
        hash_map[i] = []
```



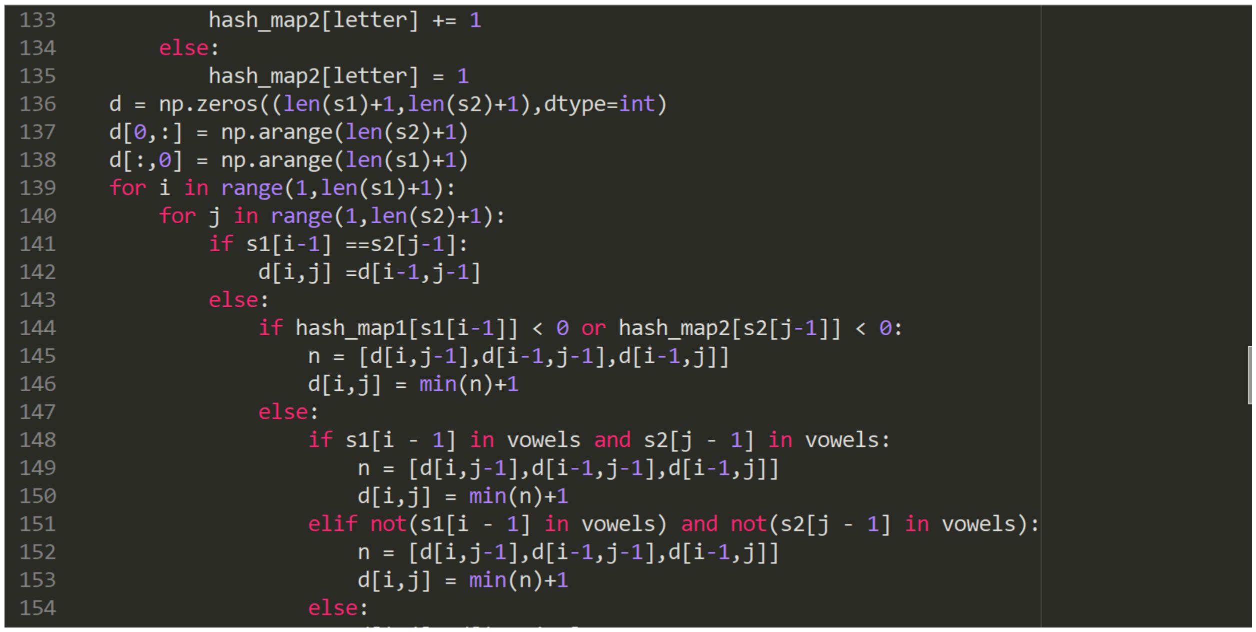
```python
    counter = len(g.al) + 1
    while counter >= 0:
        pos1 = np.random.randint(0, len(g.al))
        if len(g.al[pos1]) == 0:
            counter -= 1
        else:
            pos2 = np.random.randint(0, len(g.al[pos1]))
            if not(g.al[pos1][pos2].dest in hash_map[pos1]):
                hash_map[pos1].append(g.al[pos1][pos2].dest)
                counter -= 1
                subset.insert_edge(pos1, g.al[pos1][pos2].dest)
    return subset

#conditions() - conditions to keep the Hamiltonian path cycle,
#used in the backtrackign algorithm to see if each added edge
#is valid to keep the Hamiltonian Cycle property
#inputs - g: the graph where we are trying to find the path
#         v: the vertex we are checking to see if it is valid
#         pos: the current position of the path
#         path: the current hamiltonian path
#         counter: used to access the correct position in the
#                  path list
```



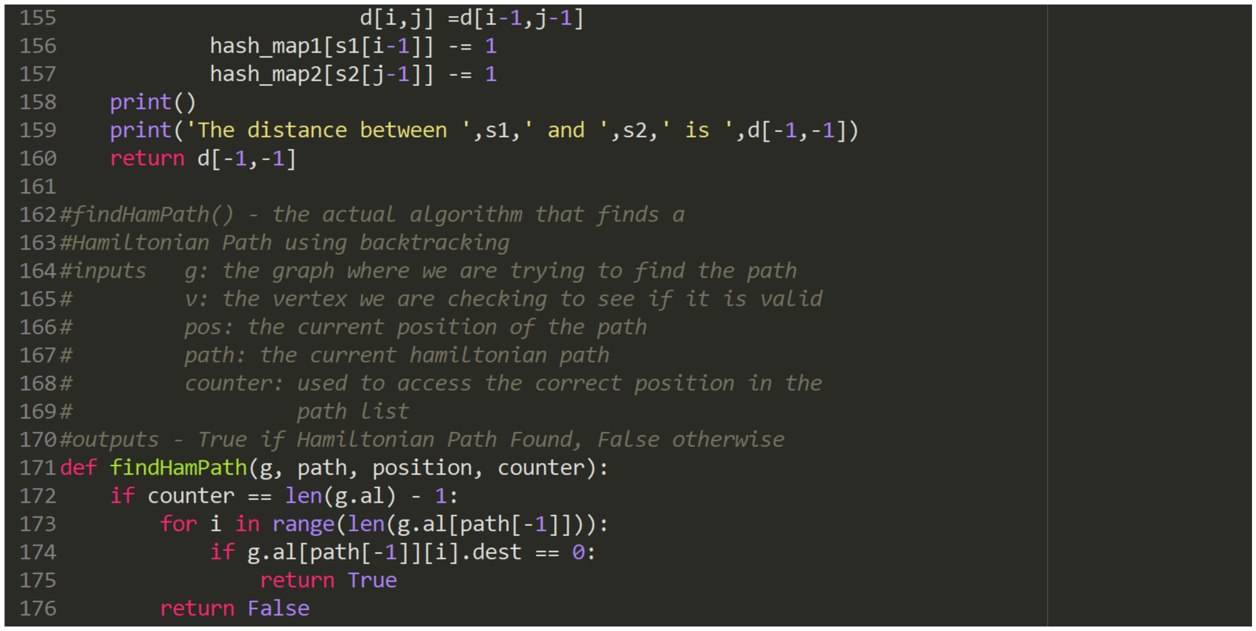
```python
#outputs - True if the vertex does not break the Hamiltonian
#          Path, False else
def conditions(g, v, pos, path, counter):
    flag = False
    for i in range(len(g.al[path[counter]])):
        if g.al[path[counter]][i].dest == v:
            flag = True
    if not flag:
        return False
    for ver in path:
        if ver == v:
            return False
    return True

#dynamicEditDistance() - initilizes the dynamic edit
#distance problem by asking the user for two words
#inputs - None
#ouputs - None
def dynamicEditDistance():
    print()
    print('Which two words would you like to use?')
    word1 = input('Word 1: ')
```



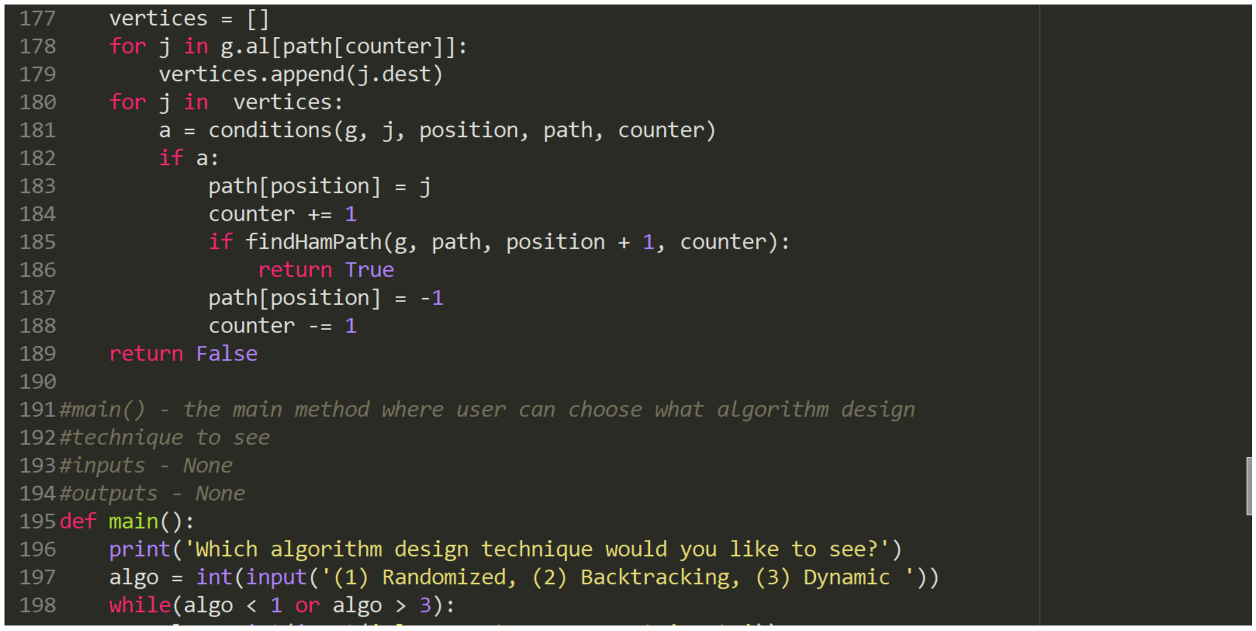
```python
    word2 = input('Word 2: ')
    editDistance(word1, word2)

#editDistance() - the same distance problem but
#only swaps characters if they are both vowels or both
#consonants
#inputs - s1: the first word
#         s2: the second word
#outputs - the distance between the two words, considering
#that different letters only count if they are both vowels
#or both consonants
def editDistance(s1,s2):
    vowels = ['a', 'e', 'i', 'o', 'u']
    hash_map1 = {}
    for letter in s1:
        if letter in hash_map1:
            hash_map1[letter] += 1
        else:
            hash_map1[letter] = 1
    hash_map2 = {}
    for letter in s2:
        if letter in hash_map2:
```



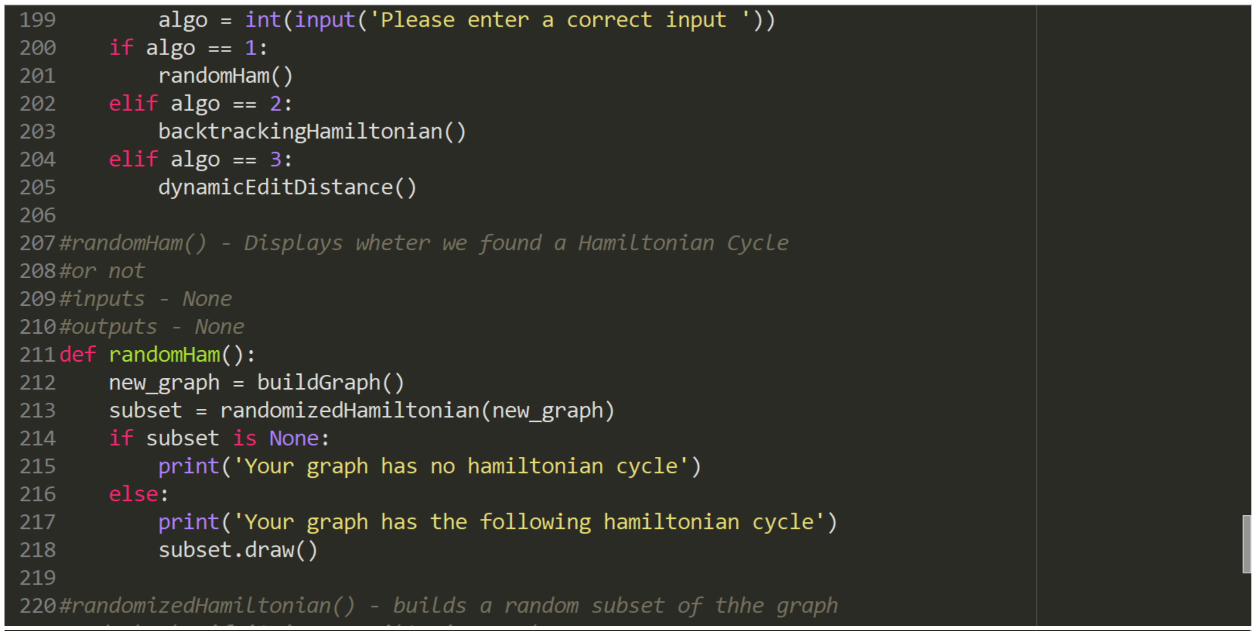
```python
            hash_map2[letter] += 1
        else:
            hash_map2[letter] = 1
    d = np.zeros((len(s1)+1,len(s2)+1),dtype=int)
    d[0,:] = np.arange(len(s2)+1)
    d[:,0] = np.arange(len(s1)+1)
    for i in range(1,len(s1)+1):
        for j in range(1,len(s2)+1):
            if s1[i-1] ==s2[j-1]:
                d[i,j] =d[i-1,j-1]
            else:
                if hash_map1[s1[i-1]] < 0 or hash_map2[s2[j-1]] < 0:
                    n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                    d[i,j] = min(n)+1
                else:
                    if s1[i - 1] in vowels and s2[j - 1] in vowels:
                        n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                        d[i,j] = min(n)+1
                    elif not(s1[i - 1] in vowels) and not(s2[j - 1] in vowels):
                        n = [d[i,j-1],d[i-1,j-1],d[i-1,j]]
                        d[i,j] = min(n)+1
                    else:
```



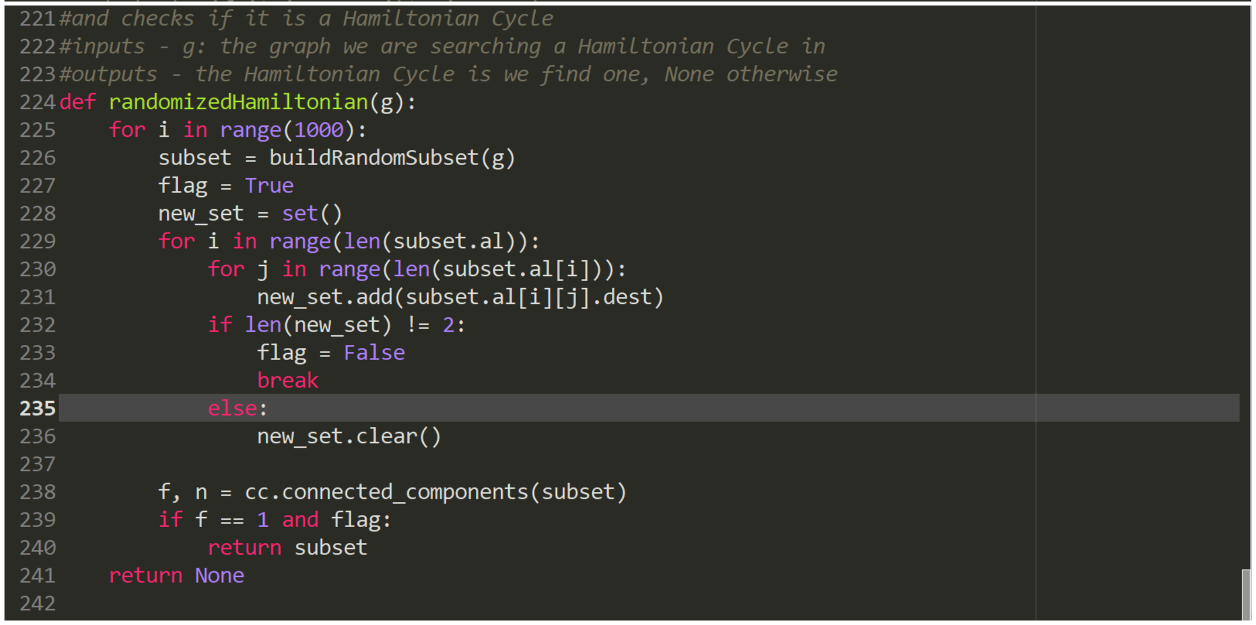
```python
                        d[i,j] =d[i-1,j-1]
            hash_map1[s1[i-1]] -= 1
            hash_map2[s2[j-1]] -= 1
    print()
    print('The distance between ',s1,' and ',s2,' is ',d[-1,-1])
    return d[-1,-1]

#findHamPath() - the actual algorithm that finds a
#Hamiltonian Path using backtracking
#inputs   g: the graph where we are trying to find the path
#         v: the vertex we are checking to see if it is valid
#         pos: the current position of the path
#         path: the current hamiltonian path
#         counter: used to access the correct position in the
#                  path list
#outputs - True if Hamiltonian Path Found, False otherwise
def findHamPath(g, path, position, counter):
    if counter == len(g.al) - 1:
        for i in range(len(g.al[path[-1]])):
            if g.al[path[-1]][i].dest == 0:
                return True
        return False
```



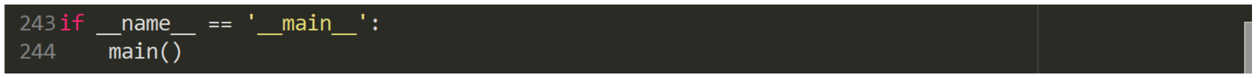
```python
    vertices = []
    for j in g.al[path[counter]]:
        vertices.append(j.dest)
    for j in  vertices:
        a = conditions(g, j, position, path, counter)
        if a:
            path[position] = j
            counter += 1
            if findHamPath(g, path, position + 1, counter):
                return True
            path[position] = -1
            counter -= 1
    return False

#main() - the main method where user can choose what algorithm design
#technique to see
#inputs - None
#outputs - None
def main():
    print('Which algorithm design technique would you like to see?')
    algo = int(input('(1) Randomized, (2) Backtracking, (3) Dynamic '))
    while(algo < 1 or algo > 3):
```



```
        algo = int(input('Please enter a correct input '))
    if algo == 1:
        randomHam()
    elif algo == 2:
        backtrackingHamiltonian()
    elif algo == 3:
        dynamicEditDistance()

#randomHam() - Displays wheter we found a Hamiltonian Cycle
#or not
#inputs - None
#outputs - None
def randomHam():
    new_graph = buildGraph()
    subset = randomizedHamiltonian(new_graph)
    if subset is None:
        print('Your graph has no hamiltonian cycle')
    else:
        print('Your graph has the following hamiltonian cycle')
        subset.draw()

#randomizedHamiltonian() - builds a random subset of thhe graph
```



```
#and checks if it is a Hamiltonian Cycle
#inputs - g: the graph we are searching a Hamiltonian Cycle in
#outputs - the Hamiltonian Cycle is we find one, None otherwise
def randomizedHamiltonian(g):
    for i in range(1000):
        subset = buildRandomSubset(g)
        flag = True
        new_set = set()
        for i in range(len(subset.al)):
            for j in range(len(subset.al[i])):
                new_set.add(subset.al[i][j].dest)
            if len(new_set) != 2:
                flag = False
                break
            else:
                new_set.clear()

        f, n = cc.connected_components(subset)
        if f == 1 and flag:
            return subset
    return None

```



```
if __name__ == '__main__':
    main()
```

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.