**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #7**

Due: December 4th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to implement three commonly used algorithm design techniques: Randomization, backtracking, and dynamic programming. Randomization involves performing a set method *n* number of times to check if, in this case, there is a Hamiltonian cycle within a graph. Backtracking involves taking one element from a set at a time through recursive calls and, at each call, checking if said element satisfies a certain condition; thus, if it does, the element can then be added back via a return statement that itself is returned to via traceback. Dynamic programming involves solving a set of smaller problems in order to solve one big problem, such as in the case of the *edit\_distance* function, which determines how many changes to a word must be made letter by letter in order to change it into another word, and does so by finding the edit distance of the two words’ substrings.

**Proposed Solution Design and Implementation**

**Part 1 (Randomized Algorithms):**

In this first part, we had to determine if there was a Hamiltonian cycle within a given graph without testing all possible subsets of the graph. To do this, we would implement a method that would make a graph out of a number of edges equal to the number of vertices, where the edges themselves are sampled randomly from the graph. Of note here is that that specific number of edges is taken because that is the minimum number of edges necessary to form a Hamiltonian cycle.

From there, we can use the *connected\_components()* method on the generated graph to determine if there is only one connected component, thus implying that all the nodes are connected to one another. If so, then we can check if the in-degree of each vertex is two, thus implying that, in a given direction, there is only one path in and one path out of that vertex. Of note here is that checking that the length of *g.al[v]* for each vertex *v*, despite being the out-degree, is just as applicable – since the graph is undirected – and can be done in O(V) time as opposed to O(V+E) time.

This is done *n* number of times until a graph which has (and is itself) a Hamiltonian cycle can be returned, or *None* can be returned otherwise. In this case, *n* = 500,000 in order to always find a cycle for graphs with up to 10 vertices.

**Part 2 (Backtracking):**

In this second part, we again had to find a Hamiltonian cycle in a graph, but this time, we would use a recursive function to find it (or determine that it didn’t exist) so as to only go through it once. To do this would normally involve considering every possible subset, but the idea here is that we take an edge, and if that edge and its vertices fit the conditions for a Hamiltonian path (as can be determined with a separate method based off of the method from Part 1), then it is returned alongside an appended list that is returned by a recursive call with that edge no longer considered. The idea here is that, should an edge ever not fit the condition, the method can not return it alongside a recursive call, and instead return just the recursive call, thereby “skipping” that edge as part of the set of edges.

The most difficult part here was determining the parameters and base cases of the method. Ultimately, I had the parameters set to the set of vertices in the graph and the set of edges in the graph (as well as the number of vertices originally within the graph, but this is more for the sake of determining the number of relevant connected components). This was so that, as the edges were considered, their source vertices could be removed, thus indicating that no further edges which shared said source vertex need be included. Should an edge ultimately not be necessary, its source vertex can be appended back into the set after the recursive call such that the return statement without that edge can make a recursive call indicating another edge from that vertex need be included.

As for the base cases, two were obvious enough: If there was only one edge in the set of edges in a given call, that edge should be returned; and if there were no edges or vertices left in a given call, then there was no more to be considered, so an empty list should be returned. That said, these simply were not enough base cases. While the function works properly for a graph that definitely has a cycle, it has no way of determining that a graph has no cycles. This would likely come from a base case that would return *None* or throw an error or some such procedure, but I ultimately couldn’t figure out how to properly implement this.

**Part 3 (Dynamic Programming):**

For this last part, we were asked to take the *edit\_distance()* method, as described previously in class, and edit it such that it only allowed the switching of letters to occur when either both were vowels or both were consonants. While initially straightforward enough, there was a bit of thinking to be had on the ramifications of such a change.

First and foremost, two lists of letters – one of vowels, the other of consonants – had to be made to determine the state of the given letters. A simple if statement was added before the final assignment of a given spot in the matrix that would set said spot’s value to the minimum of the three values north, east, and northeast of it.

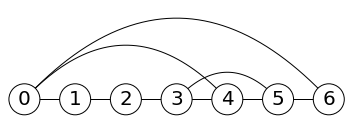
As for what to do when two given letters weren’t both vowels or consonants, the answer came soon enough: Simply take the maximum of the three adjacent previous values rather than the minimum, as since switches (1 change) couldn’t occur, it is implied instead that a change from vowel to consonant, or vice versa, would take a removal and insertion (2 changes), or vice versa.

**Experimental Results**

**Part 1:**

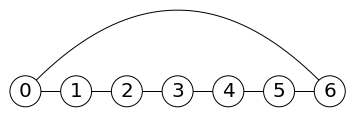
I had the program test the method with a graph that definitely had a Hamiltonian cycle (as well as additional random edges) and a graph composed solely of random edges that most likely did not. If a cycle was found, the program would print the number of trials needed to find it and draw it. Otherwise, it would simply print “Hamiltonian cycle not found.”

**Inputted Graph w/ Hamiltonian Cycle:**

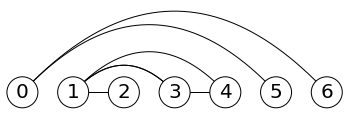
****

**Result:**





**Inputted Graph w/o Hamiltonian Cycle:**



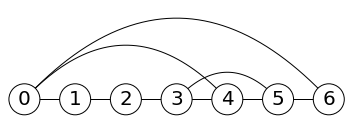
**Result:**



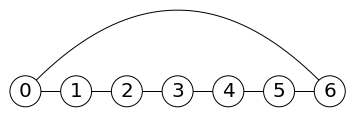
**Part 2:**

This method was tested alongside the *hamiltonian\_randomized()* method, and thus should have theoretically returned the same outputs, but as mentioned previously, it simply could not express that a graph did not have a Hamiltonian cycle, instead returning a Hamiltonian cycle that covers only part of the graph.

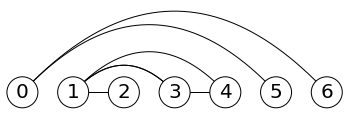
**Inputted Graph w/ Hamiltonian Cycle:**

****

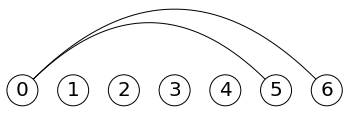
**Result:**



**Inputted Graph w/o Hamiltonian Cycle:**



**Result:**



**Part 3:**

I had the program print out the edit distances of several pairs of words with arbitrary lengths specifically so I could demonstrate in this paper what those changes would be so as to show that the numbers produced are correct. Following each printed statement, I will write down one example of incremental changes to be made such that the first word becomes the second word, with each “➝” in between indicating a change.



sand➝sond➝sound



corn➝con➝coan➝coat



feet➝feel



trash➝rash➝urash➝unrash➝unirash➝unicash➝unicosh➝unicorh➝unicorn



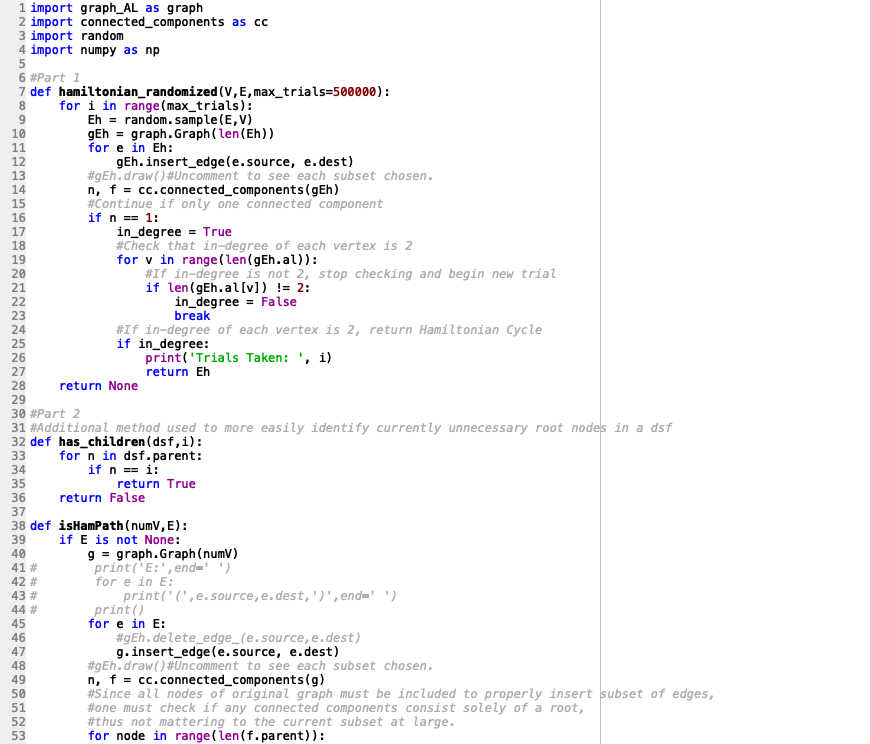
alabama➝malabama➝milabama➝misabama➝missabama➝missibama➝missisama➝

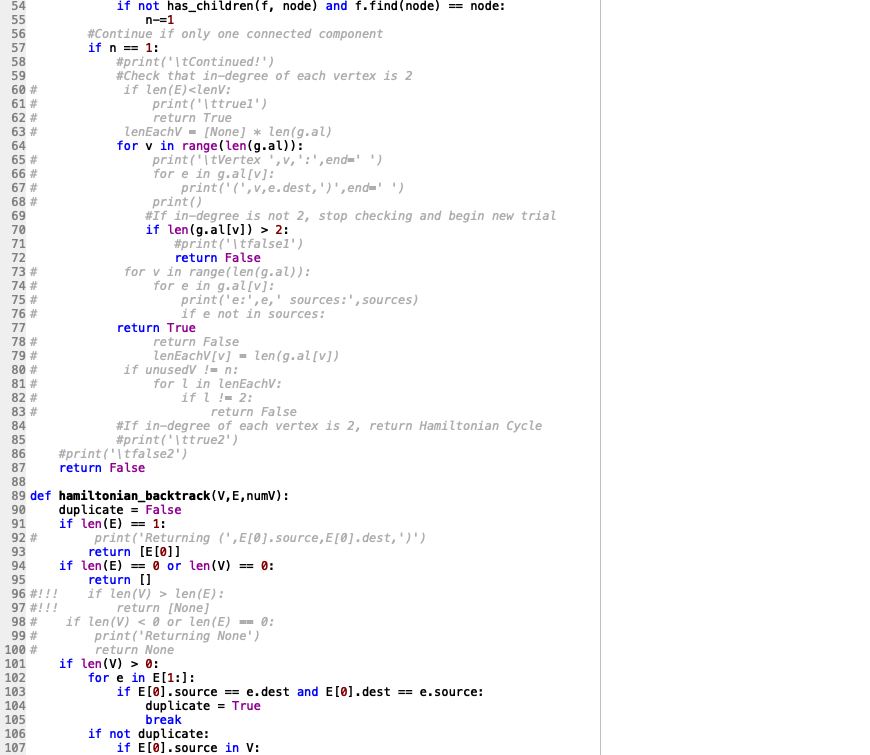
mississama➝mississima➝mississipa➝mississippa➝mississippi

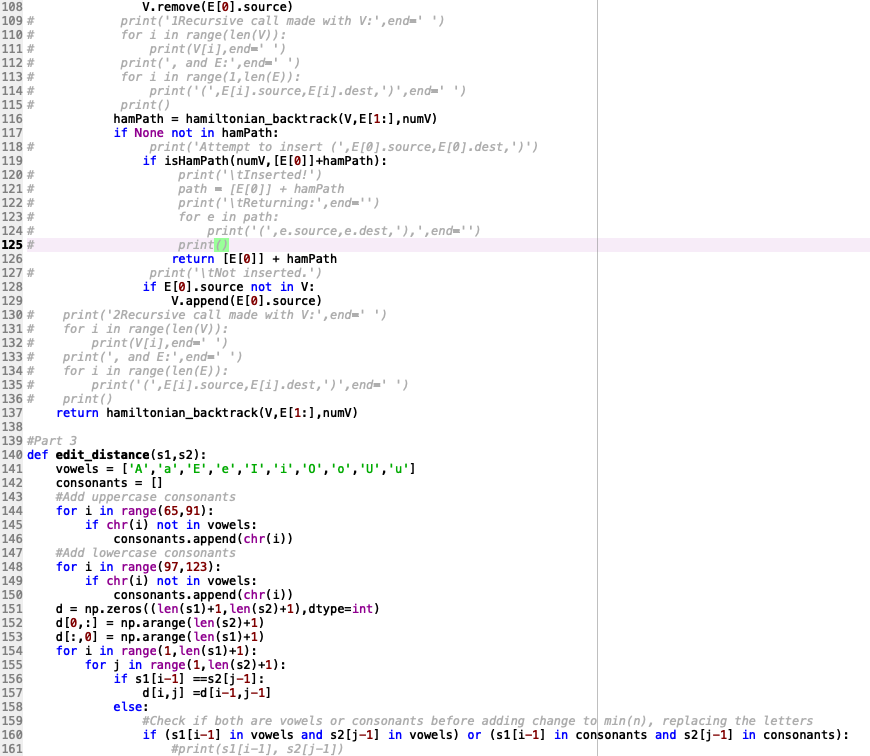
**Conclusion**

From this lab, I learned how methods involving these techniques can be utilized to solve time-consuming tasks. Rather than consider every possible subset of a graph, you can simply have a method test so many random subsets of that graph, and, within so many attempts, find one that has a Hamiltonian cycle; alternatively, you can have a method store necessary values on its own via recursion and the inherent stack used as its running so as to have it explore a graph from vertex to vertex and head back to a previously found vertex to try a different path if the one first explored went awry; and as for the edit distance of two given words, I learned that you can represent a removal and insertion (or vice versa) instead of a replacement by taking the largest number of possible changes made to the previous substrings rather than the smallest.

**Appendix**











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.