CS2302 Data Structures

Fall 2019

Lab Report 7

Author: Bryan Ramos

Due: December 6th 2019

Professor: Dr. Olac Fuentes

TA: Anindita Nath

Introduction

This is the last laboratory assignment of the course. The final thing we learned in the lecture and in the Zybooks readings about four kinds of algorithms: randomization, greedy, backtracking and dynamic programming. The lab provides practice with randomization and backtracking to solve an NP-complete problem of finding a Hamiltonian cycle. A Hamiltonian cycle in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle is a Hamiltonian path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian path.

In addition, the lab allows for deliberate practice with dynamic programming to modify the edit distance of two strings algorithm we discussed in the course.

Proposed Solution Design and Implementation

Before writing any other code for the main lab file, I imported some of the files either from the class webpage or what we wrote in class that I felt would be necessary to complete this lab assignment, these include: adjacency list, edge list, and connected components using DSF. I also included the code for the adjacency matrix so that I did not have to modify the adjacency list or edge list files.

Next, before writing any functions, in order to split up my output for each of the algorithms, I created a menu so that each of the algorithms runs separately. I solve the lab instruction in order. First, the randomization problem. To find whether a Hamiltonian Cycle exists using randomization, my function takes two parameters: a number for trials to run, and an adjacency list graph representation.

The pseudocode provided by Dr. Fuentes in the lab pdf clarified how the algorithm had to be designed. I also did some further research to understand how a Hamiltonian cycle algorithm works by analyzing articles on GeeksForGeeks. Although I passed an adjacency list, in order to provide efficiency and for ease of use, I converted by adjacency list to

another representation, an edge list. Next, since the randomization algorithm must go through several trials, I used a for loop to iterate.

For each iteration of the for loop based on the number of trials, an adjacency list graph representation is created, and then converted to an edge list, again, for ease of use. Edges will be randomly chosen and added to the new graph, until the number of edges matches the number of vertices.

Next thing, the algorithm must check if the graph has one connected component and if each vertex has an in-degree of two. If the conditions match the specifications, the graph is returned, otherwise its necessary to check randomly for trials for a Hamiltonian cycle. Again, because of the pseudocode provided, I was quickly able to solve this algorithm.

Second is trying to solve the Hamiltonian cycle problem using backtracking. The design of the solution is like the previous one but utilizes recursion to solve with backtracking. As before, the adjacency list representation is converted into an edge list for ease of use. For the recursive function to work, two parameters are passed: the graph originally passed to the function and an empty graph. In the recursive function, an edge from the graph will be inserted into the empty graph. Like with the randomization algorithm, now its necessary to look for a Hamiltonian cycle.

Let's suggest a graph is built. To tackle the Hamiltonian cycle problem, the edges will be checked to make sure that the connected components is 1 and that all vertices have an in_degree of 2. For finding the in degree I ended up using the code from the quiz we had a few weeks back on in_degrees and out_degrees. Anyway, if the conditions are true, we are done, the recursion ends, otherwise, the algorithm must go through all possible combinations to find combinations.

Edit distance was one of the final things we covered too for a brief period of time. Like I mentioned before, since the instructions call for editing the edit distance algorithm, I imported the code from the one we

wrote and learned in class. The instruction state that the new edit distance algorithm should “allow replacements only in the case where the characters being interchanged are both vowels, or both consonants. For example, ‘a’ can be replaced by ‘e’ but not by ‘s’ and ‘t’ can be replaced by ‘w’ but not by ‘u.’

Anyway, the first thing necessary to do is to build a 2d matrix, or 2d list, that is m by n (m x n), that m the length of the first string, by n, the length of the second string. Next, two python lists are created, one containing all the consonant letters of the alphabet and the other containing vowels. My code puts in a provision where all of the inputted strings are changed to lowercase letters, just in case that input to the function would mean string 1 would contain lowercase letters, string 2 would contain uppercase letters, then the function will still work correctly. Now, for every character my algorithm checks, the upper diagonal is only brought in if both characters checked are equal. If not, the algorithm must check if both are vowels or if both are consonants and finds the minimum of the left, top or diagonal, adding one. If not, it only allows the min of the top or left, adding one.

Experimental Results

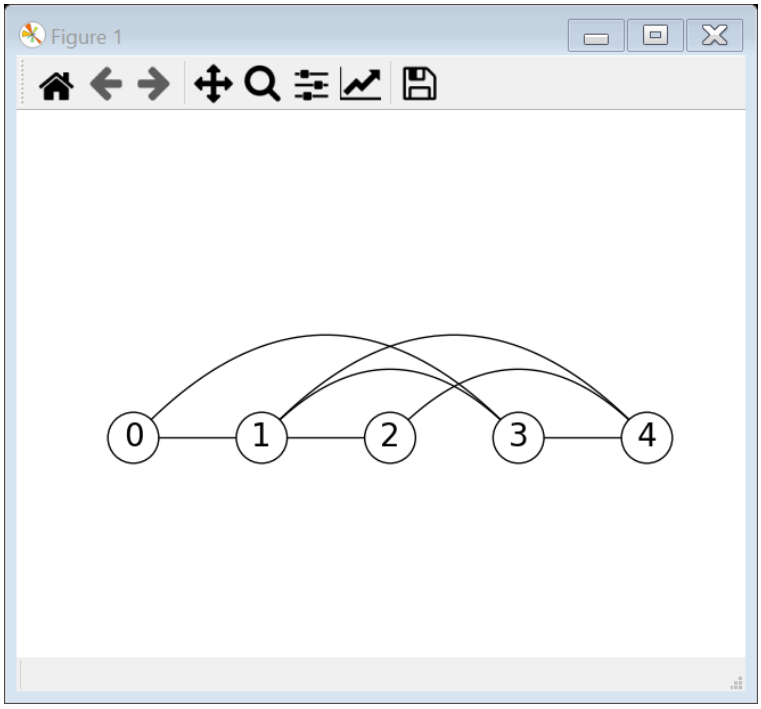
I provide a menu to allow the user to split each function up so the methods and their running is individual.

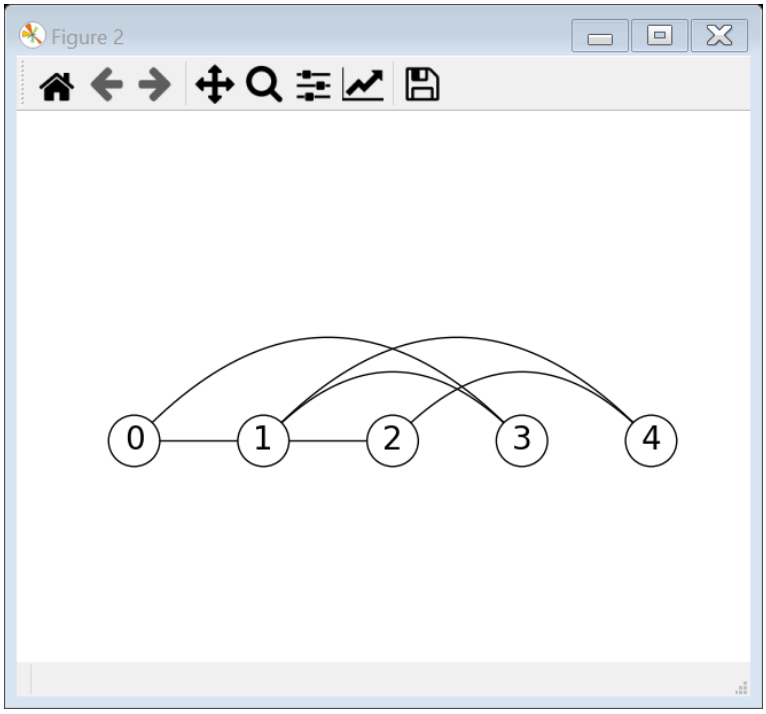
```
In [22]: runfile('C:/Users/Bryan Ramos/Documents/CS2302/lab7/
lab7.py', wdir='C:/Users/Bryan Ramos/Documents/CS2302/lab7')
Reloaded modules: graph_AL, cc, dsf
Options
1. Randomized Hamiltonian
2. Backtracking
3. Dynamic Programming

Choose an option:
```

On the screenshot below of the Python console, the randomization algorithm returns the results for two checks on two graphs to check for a Hamiltonian cycle. The graph should be a Hamiltonian, the second graph is the same as the first but adding one edge, breaking the chance of the

second graph from being a Hamiltonian cycle. So the output should show that a Hamiltonian cycle should be found for the first graph but not the second.





For the backtracking algorithm, although the implementation is slightly different, the solutions should be the same, which they are.

```
Options
1. Randomized Hamiltonian
2. Backtracking
3. Dynamic Programming

Choose an option: 2
0 1 1
0 3 1
1 0 1
1 2 1
1 4 1
1 3 1
2 1 1
2 4 1
3 1 1
3 0 1
3 4 1
4 2 1
4 1 1
4 3 1
Solution exists: Hamiltonian Cycle Found
Runtime: 0.16628623008728027
0 1 1
0 3 1
1 0 1
1 2 1
1 4 1
1 3 1
2 1 1
2 4 1
3 1 1
3 0 1
4 2 1
4 1 1
Could not find hamiltonian cycle!
Runtime: 0.24663829803466797
```

Finally, testing the modified edit distance function with some samples strings passed.

```
Options
1. Randomized Hamiltonian
2. Backtracking
3. Dynamic Programming

Choose an option: 3
Word 1 money  Word2 miners
[[0 1 2 3 4 5 6]
 [1 0 1 2 3 4 5]
 [2 1 1 2 3 4 5]
 [3 2 2 1 2 3 4]
 [4 3 3 2 1 2 3]
 [5 4 4 3 2 3 4]]
Runtime: 0.004403829574584961
Word 1 aei  Word2 iou
[[0 1 2 3]
 [1 1 2 3]
 [2 2 2 3]
 [3 2 3 3]]
Runtime: 0.0
Word 1 aei  Word2 bcd
[[0 1 2 3]
 [1 2 3 4]
 [2 3 4 5]
 [3 4 5 6]]
Runtime: 0.0
Word 1 utep  Word2 utsa
[[0 1 2 3 4]
 [1 0 1 2 3]
 [2 1 0 1 2]
 [3 2 1 2 2]
 [4 3 2 2 3]]
Runtime: 0.0009942054748535156
```

Conclusion

In class we learned about four algorithms, in the lab three were implemented: randomization, dynamic programming and backtracking. We discovered in class that each of these algorithms are ideal for different situations. This lab shows that the Hamiltonian cycle problem can be solved using randomization, and even with backtracking (recursion) as the correct output shows.

Going back to the beginning of the course, sometimes recursion can be very slow, especially when dealing with lots of data, the Hamiltonian cycle problem is no different. Let's say someone wants to test a large graph implementation, then backtracking will be slower than randomization.

The edit distance algorithm that we learned in class was straightforward. I recognized that edit distance is an algorithm used by spell checkers to determine the proximity of an incorrectly spelled word, to the way its correctly spelled in a dictionary.

Appendix

```
'''
Course: CS2302 Data Structures Fall 2019
Author: Bryan Ramos [88760110]
Assignment: Lab 7
Instructor: Dr. Olac Fuentes
TA: Anindita Nath
Last Modified: December 5th 2019
Purpose: Implement different kinds of algorithms: randomization,
backtracking
and dynamic programming.
'''

# imports
import graph_AL as graph
import numpy as np
from cc import connected_components
import time

# using my code from the quiz for in degree of vertex v
def in_degree(g, v):
    d = 0
    for vert in range(len(g.al)):
        for edge in g.al[vert]:
            if edge.dest == v:
                d += 1
    return d

def randomizedHamiltonian(G, trials = 1000):
    edgeList = G.as_EL()

    for t in range(trials):
        # inherit attributes from G
        g = graph.Graph(len(G.al), weighted=G.weighted,
directed=G.directed)

        # convert to edge list
        g = g.as_EL()

        edges = edgeList.el[:] # reference to copy of the list
        for vertex in range(g.vertices):
            g.el.append(edges.pop(np.random.randint(len(edges))))
```

```
        # convert back to adjacency list
        g = g.as_AL()
        # check for connected components
        c, s = connected_components(g)

        # if graph has one connected component and the in-degree of every
vertex in g is 2,
        # forms hamiltonian cycle

        # if graph has one connected component
        if c == 1:
            check_degree2 = True
            # check if the in-degree of every vertex in g is 2
            for v in range(len(g.al)):
                indegree = in_degree(g, v)
                # if the in_degree of just one vertex is not 2, then it is
not a hamiltonian cycle
                # because every vertex in g must have an in degree of 2
                if indegree != 2:
                    check_degree2 = False

            # if there is a hamiltonian cycle return g
            if check_degree2:
                return g

    # no hamiltonian cycle was found
    return None

def printCycleInfo(cycle, g):
    if cycle is not None:
        print("Solution exists: Hamiltonian Cycle Found")
        g.draw()
    else:
        g.draw()
        print("Could not find hamiltonian cycle!")

# check if every vertex has in degree 2
# parameters: list, edge list
def backtracking(g1, g2):
    # out of edges = stop
    if len(g2.el) == g2.vertices:
        adjacencyList = g2.as_AL()
        c, s = connected_components(adjacencyList)
        if c == 1:
            for i in range(len(adjacencyList.al)):
                # if the in_degree of just one vertex is not 2, then it is
not a hamiltonian cycle
                # because every vertex in g must have an in degree of 2
                if in_degree(adjacencyList, i) != 2:
```

```
                    return None
            return adjacencyList
        return None
    if len(g1) == 0:
        return None
    else:
        next = g1[0]
        g2.el = g2.el + [next]
        answer = backtracking(g1[1:], g2)
        if answer is not None:
            return answer
        g2.el.remove(next)
        answer = backtracking(g1[1:], g2)
        return answer

# backtracking algorithm
def backtrackingHamiltonian(G):
    # turn graph into edge list
    edgeList = G.as_EL()

    for e in edgeList.el:
        print(e.source, e.dest, e.weight)

    g = graph.Graph(len(G.al), weighted=G.weighted, directed=G.directed)
    return backtracking(edgeList.el, g.as_EL())

# dynamic programming
def editDistance(word1, word2):
    m = np.zeros((len(word1)+1, len(word2)+1), dtype=int)
    consonants =
['b','c','d','f','g','h','j','k','l','m','n','p','q','r','s','t','v','w','x
','z']
    vowels = ['a','e','i','o','u']

    print("Word 1 {}  Word2 {}".format(word1, word2))

    word1.lower()
    word2.lower()

    for i in range(len(m)):
        m[i][0] = i

    for i in range(len(m[0])):
        m[0][i] = i

    for i in range(1, len(m)):
        for j in range(1, len(m[i])):
            # if chars are equivalent
            if word1[i-1] == word2[j-1]:
                m[i][j] = m[i-1][j-1]
```

```
            else:
                # if both are consonants and vowels, find the min + 1
                if (word1[i-1] in consonants and word2[j-1] in consonants)
or (word1[i-1] in vowels and word2[j-1] in vowels):
                    m[i][j] = min(m[i-1][j], m[i][j-1], m[i-1][j-1]) + 1
                else:
                # otherwise both are not consonants or vowels
                # not possible to replace (upper left)
                    m[i][j] = min(m[i-1][j], m[i][j-1]) + 1
    print(m)

if __name__ == "__main__":

    # test for hamiltonian cycle - should have hamiltonian cycle
    g1 = graph.Graph(5, weighted=False, directed=False)
    g1.insert_edge(0,1)
    g1.insert_edge(1,2)
    g1.insert_edge(2,4)
    g1.insert_edge(1,4)
    g1.insert_edge(1,3)
    g1.insert_edge(0,3)
    g1.insert_edge(3,4)

    # test for no hamiltonian cycle
    g2 = graph.Graph(5, weighted=False, directed=False)
    g2.insert_edge(0,1)
    g2.insert_edge(1,2)
    g2.insert_edge(2,4)
    g2.insert_edge(1,4)
    g2.insert_edge(1,3)
    g2.insert_edge(0,3)

    choice = 0

    while choice < 1 or choice > 3:
        print("Options\n1. Randomized Hamiltonian\n2. Backtracking\n3.
Dynamic Programming")
        choice = int(input("Choose an option: "))

    # randomized hamiltonian
    if choice == 1:

        start = time.time()
        cycle = randomizedHamiltonian(g1)
        printCycleInfo(cycle, g1)
        end = time.time()
        print("Runtime: {}".format(end - start))

        start = time.time()
        cycle = randomizedHamiltonian(g2)
```

```
        printCycleInfo(cycle, g2)
        end = time.time()
        print("Runtime: {}".format(end - start))

    # backtracking hamiltonian
    if choice == 2:

        start = time.time()
        cycle = backtrackingHamiltonian(g1)
        printCycleInfo(cycle, g1)
        end = time.time()
        print("Runtime: {}".format(end - start))

        start = time.time()
        cycle = backtrackingHamiltonian(g2)
        printCycleInfo(cycle, g2)
        end = time.time()
        print("Runtime: {}".format(end - start))

    # dynamic programming
    if choice == 3:

        start = time.time()
        editDistance("money", "miners")
        end = time.time()
        print("Runtime: {}".format(end - start))

        start = time.time()
        editDistance("aei", "iou")
        end = time.time()
        print("Runtime: {}".format(end - start))

        start = time.time()
        editDistance("aei", "bcd")
        end = time.time()
        print("Runtime: {}".format(end - start))

        start = time.time()
        editDistance("utep", "utsa")
        end = time.time()
        print("Runtime: {}".format(end - start))
```

I, Bryan Ramos, 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.