November 25, 2020 Clarke A. Homan

MIT 6.0002 PS 2 WRITE UP

1. INTRODUCTION

Problem Set 2 for the MIT 6.0002 online course has the student complete a program provided by the course that deals with the use of mathematical graphs and a depth first search algorithm. The goal of the of the program is to compute best paths between MIT buildings subject to distance constraints. The two distance constraints are total distance between the starting and ending building (nodes) and the total outside distance between the two buildings. A text datafile describes each building as building numbers and each line within the datafile describe two building numbers and their distance between them in both total and outside distances.

The program also introduces the student to the use of the Python "Unittest" built-in testing mechanism. To understand what is going on with the Unittest mechanism as it pertains to the program, the student will need to research it on their own.

This program utilizes Python 3 (I am currently using Python 3.8.2).

2. SUMMARY DESIGN APPROACHES

Although much of the PS2 program that was provided by the course was completed, several modules were left to the student to complete. Their design descriptions are listed below.

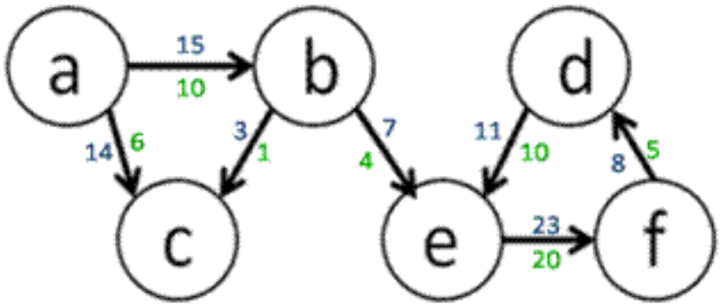
A. WeightedEdge and Digraph Classes in graph.py

i. WeightedEdge Class
The WeightedEdge class is a subclass of the Edge class predefined in graph.py program file. What needed to be completed for the class is the "__init__" constructor along with the get_total_distance and get_outdoor_distance methods which return those values stored within the class.. Also, the "__str__" stringification method that will be used to print out the edge when needed.

ii. Digraph Class
The add_node() method adds the supplied building node to the list of nodes identified within the Digraph. This function tests for the duplication of a previously entered edge into the Digraph's list of edges.

The add_edge() method adds a supplied edge (extracted from the datafile) to the Digraphs dictionary of edges, where the dictionary primary key is a source node. Thus, each Digraph's dictionary entry is keyed to a common source node that is

associated with a list of edges that the source node and some destination node is connected to. For example, the Digraph's dictionary "edges" entry for source node "a", contains two edges in a list (as the dictionary element's data), that list a->b and "a->c" edges. Each edge indicates the total distance between nodes (blue number) and outside distance between nodes (green number)



B. Load_map function in ps2.py

The load_map() function opens a text file, whose name is supplied as an argument, and proceeds to extract the contents of the file, line by line until "end of file" is determined. Each line is parsed to the source and destination bldg. numbers and their distances. When the source and destination bldg. numbers for the line is identified, nodes are created for each. Before they are added to the Digraph, the Digraph is texted to see if the building nodes are already entered. If either is not the case, their nodes are added to the Digraph. Then an edge is created for the Digraph's edge dictionary, keyed to the source node. The edge is then added to the Digraph's edge dictionary element's edge list.

An extra design feature added to the load_map() function is the support of commenting within the datafile. Comments can be added per line to a datafile by preceding the line with a '#' character. This was not requested by the problem set assignment, but I felt it was a nice add.

Once completed, the datafile is then closed.

C. Get_best_path and directed_dfs functions in ps2.py

i. **Get_best_path()** is the "work horse" function. Its mission is to determine from a starting node, the "best" path to the "ending" node, supplied as parameters, using a depth first search algorithm. This search is constrained by having its total and outside distances (cumulative from the starting to ending bldg distances thru the in between bldgs. distances).

The challenge is that although a depth first search algorithm will indeed find a minimum path from the source (starting) to (end) ending node given these constraints, it by itself will not necessarily determine the best (lowest) possible path, because it by itself doesn't remember what it discovered before. So, it may

rediscover a path it already discovered. The get_best_path() function is enhanced by providing the path it already discovered along with the distance that previous discovery discovered. If it can beat what it found already, it offers that recommendation as the new "best" path.

Get_best_path recursively iterates "down" a path trying to establish a set of connected nodes starting from the start bldg that "finds" the ending node (bldg) whose distances do not exceed the maximum and maximum outside distances restrictions.

ii. Directed_dfs() is called by the main program and calls get_best_path(). It receives the starting and ending building numbers and the distance restrictions (both total and outdoors). It initially calls the get_best_path() function to get a 1st candidate "best" path. If get_best_path() can discover a path between the source node and the destination node whose sum of intermediary distances do not exceed the total and outside distances, then it recursively promotes the path as a "candidate" best path to "directed_dfs". Directed_dfs then records the candidate path recommendation, and then iterates get_best_path() again, but supplying what it previously found as a path recommendation. Get_best_path() then attempts to find yet another path that does not include the path elements it previously found, in the hope that it can find another path. Directed_dfs() then compares the distances from the previous call to get_best_path with what is got subsequent iterations of get_best_path(). When get_best_path() exhausts the path possibilities and returns "None", then directed_dfs() then returns the best path it encountered to the calling program.

3. RESULTS

The results of the program did accomplish what was required. The program reads the datafile(s) correctly and calculates best paths, provided the end point nodes and distance requirements specified.

When the "Unittest" code is executed, two test cases "fail":

```
======================================================================
FAIL: test_path_multi_step (__main__.Ps2Test)
----------------------------------------------------------------------
AssertionError: Lists differ: ['2', '3', '7', '9'] != ['2', '4', '10', '3', '1', '5', '7', '9']


======================================================================
FAIL: test_path_multi_step_no_outdoors (__main__.Ps2Test)
----------------------------------------------------------------------
```

AssertionError: Lists differ: ['2', '4', '10', '13', '9'] != ['2', '6', '8', '16', '26', '36', '34', '24', ''9']

In the first test case, “multistep”, the program when executed with a “wide” distance requirement (99999), differs from expected by the side jog of adding nodes 4, 10 and 1, 5 to the spec. If the same code is executed (without the unittest mechanism) directly, with the distance requirements of “115” and “200” for total and outdoor, then the program does indeed yield the expected path of [‘2’, ‘4’, ‘7’, ‘9’].

In the second test case, “no outdoors”, the program when executed with “wide” distance requirements, differs from the expected. Here the divergence is much larger, as the suggested best path doesn’t resemble the expected at all. If the same code is executed (without the unittest mechanism) directly, with the distance requirements of “155” and “0” for total and outdoor, then the program improves to yield a best path of ['2', '4', '10', '13', '9'].

In both cases, by relaxing the ‘total distance” requirements at the start, the program suggests a progressively less optimal best path recommendation. An improvement could be made at the function directed_dfs() whereby progressively tighter total distance requirements are provided to subsequent get_best_path() invocations until it returns “None” which then indicates that no path exists with those set of requirements. Then directed_dfs would then return the last best path it had recorded in the past. Although I did not modify the code to do this, I am confident that if I did, the program would perform correctly as expected.

4. CONCLUSIONS

I found this exercise interesting and I learned from the experiment. Perhaps the design of the code was not optimal, but in the end, the program performed as it was expected to a large extent. One thing I like in particular with respect to the design of the code is that the results provided will differ based upon the distance requirements provided. My suspicion is that the code would execute less when given lax distance requirements, still meeting those requirements it was provided, but when given more stringent distance requirements, it look more. Thus, I would characterize the implementation as “lazy” but still correct when needed. Perhaps a reflection on the programmer’s nature. ☺