FIT2004 S1/2019: Assignment 4 Questions

THIS PRAC IS ASSESSED! (7.5 Marks)

Last updated: 15th May 2019, 14:40:00 AEST

DEADLINE: Friday 31st May 2019 23:55:00 AEST

LATE SUBMISSION PENALTY: 20% penalty per day.
Submitting after 5 days of deadline, you will get 0.
For special consideration, please complete and send the *in-semester special consideration* form with appropriate supporting document to `fit2004.allcampuses-x@monash.edu`.

PROGRAMMING CRITERIA: It is required that you implement this exercise strictly using **Python programming language** (version should not be less than 3.5). This practical work will be marked according to the functionality. Solutions that are not within time/space complexity requirement would be penalized. As part of a good programming practice, your program would need to adhere to a reasonable coding standard such as function documentations. If you do not follow the coding standard or program design mentioned in the specification, there is a penalty of 20% of your total marks. Lastly, you would need to include an analysis report of your solution – outlining the algorithm of your program and a detailed complexity analysis of your solution.

SUBMISSION REQUIREMENT: You will submit your program within a zipped file named `studentId_A4.zip`, e.g. if your student id is XXXX, the zipped file is `XXXX_A4.zip`. Moreover, your zipped file should be ONLY in `.zip` extension – containing your Python program file (named `roadPath.py`) and your analysis report PDF file (named `Report_A4.pdf`). The PDF file must give the details of your solution along with the **worst-case** space and time complexity. Penalties will be applied if the PDF file is missing. The zipped file has to be submitted on Moodle before the deadline. Please note that a draft submission is considered as not-submitted and will not be marked.

PLAGIARISM: The assignments will be checked for plagiarism using an advanced plagiarism detector. Last year, many students were detected by the plagiarism detector and almost all got zero mark for the assignment and, as a result, many failed the unit. "Helping" others is NOT ACCEPTED. Please do not share your solutions partially or/and completely to others. If someone asks you for help, ask them to visit us during consultation hours for help.

©

PLEASE READ THE ENTIRE ASSIGNMENT SPECIFICATION CLEARLY BEFORE PROCEEDING TO IMPLEMENTATION

In the assignment, you will develop a road routing program – one that is similar to navigational systems such as the one used Google map and Uber. You would do this according to the following 3 assignment tasks.

As a start before that, you would require to build a **Graph** class. The structure of the class is similar to the one discussed in **Tutorial 9 Problem 1** – a **positively weighted directed graph** implemented with the **adjacency list** representation. Thus, implement your **Graph** class with the following initialization `__init__(self)`.

Your **graph object** would be used to represent a road network, which would be read from a list of edges stored within the input file named `basicGraph.txt`. In the input file, each line corresponds to a weighted directed edge and contains 3 space-separated values:

- vertex u represented as an integer.
- vertex v represented as an integer.
- time w to travel from vertex u to vertex v (in minutes) as the weight of the edge represented as a float. You can assume the travel time are positive values only.

Below is an example, showing a part edge list file `basicGraph.txt`:

```
10 221 5
211 22 2
110 212 7.5
```

For example, the 3rd row shows that there is a road connecting location 110 to location 212 that would require a travel time of 7.5 minutes. This would be represented in your *Graph* as an edge going from vertex 110 to vertex 212 with a weight of 7.5.

The input file is read using the function within your **Graph** class called `buildGraph(self, filename_roads)` where `filename_roads` denotes the filename of the road network. For example, invoking `buildGraph(self, "basicGraph.txt")` would populate the **Graph** object with vertices and edges according to the edge list present in `basicGraph.txt`.

Important: You are to assume that the vertex ID is continuous, starting from vertex 0 up to vertex k where k is the largest vertex in the edge list stored within the input file named `basicGraph.txt`. This would generate an adjacency list graph representation with $k + 1$ vertices, with the vertex ID from 0 to k.

You are free to add any instance variables or attributes that you deemed necessary to the **Graph**, **Vertex** and **Edge** class. Do however be mindful of the space complexity of your implementation; it needs to be within the space complexity of $O(E + V)$ where V is the total number of points/ nodes/ vertices in the road network; and E is the total number of roads/ edges in the road network.

1 Task 1: Quickest Path

In this task, you would need to find the quickest path from a given starting location `u` to a destination `v`. These locations are vertices within the constructed **Graph**, with respect to the road network in the `basicGraph.txt` input file.

To accomplish this, you would need to create a function named `quickestPath(self, source, target)` within your **Graph** class. It would accept the following 2 arguments:

- `source` which denotes the starting point of the travel.
- `target` which denotes the destination point of the travel.
- Note that both points are valid vertices in the graph.

The function would then return 2 values within a tuple:

- `list` containing all nodes in the order of the quickest path traversal from the `source` to the `target`.
- `time` storing the total time require from reach the `target` from the `source`.

For example, invoking the function `my_graph.quickestPath("0", "15")` returns the tuple `([0,2,10,1,3,4,5,15], 120.0)`.

- `[0,2,10,1,3,4,5,15]` which denotes the traversal from source 0 to target 15, going through the path of `0 -> 2 -> 10 -> 1 -> 3 -> 4 -> 5 -> 15`.
- 120.0 which denotes the total travel time (in minutes) from source 0 to target 15.
- Note that there can be multiple quickest paths going from source 0 to target 15. Thus, the solution is not unique and it is fine for your program to return any of them.

If a path does not exist between source 0 and target 15 (i.e. vertex 15 is not reachable from vertex 0), then your function would return `[[],-1]` – an empty path `[]` with a traversal time of -1.

Your function should run within the time complexity of $O(E \log V)$ where V is the total number of points/ nodes/ vertices in the road network; and E is the total number of roads/ edges in the road network. The space complexity remains the same as the space complexity of your input graph $O(E + V)$.

2 Task 2: Safe Quickest Path

Just like in the real world, certain paths could undesirable to travel – having tolls, under roadworks or congested with heavy traffic. For this assignment, we would add the following to our road networks and your constructed graph:

- Red light cameras as various locations/ points/ nodes/ vertices.
  Information about which location having red light cameras are stored within the input file `camera.txt` as can be seen below.

  ```
  211
  110
  ```

  Here, we observe that vertex 211 and vertex 110 do contain red light cameras.
  You are free to assume that the vertices listed in the file are all valid vertices.
  The rest of the unmentioned vertices are without red light cameras.

- Tolls at various roads/ edges.
  Information about which road having tolls are stored within the input file `toll.txt` as can be seen below.

  ```
  211 22
  110 212
  ```

  Here, we observe that the edge from vertex 211 to vertex 22 do contain a toll. Likewise, the same observation can be made for the edge from vertex 110 to vertex 212.
  You are free to assume that the vertices listed in the file are all valid vertices. The rest of the unmentioned edges are without tolls.

It is up to your ingenuity on how would you represent these additional elements into your **Graph** data structure. Do however ensure that the space complexity remains at $O(V + E)$.

To accomplish this, you would need to create a function named `augmentGraph(self, filename_camera, filename_toll)` within your **Graph** class. It would accept the following 2 arguments:

- `filename_camera` which denotes filename of the text file containing the list of red light cameras.
- `filename_toll` which denotes the filename of the text file containing the list of tolls.

For our assignment, we define a **safe path** as the following – A **safe path** is a path in which (1) none of the vertices within the path contains a red light camera; and (2) none of the edges along the path are toll roads.

Thus, we would want to extend our quickest path solution from Task 1 to account for the additional constraint of a safe path. You would need to create the function named `quickestSafePath(self, source, target)` within your **Graph** class. It would accept the following 2 arguments:

- `source` which denotes the starting point of the travel.
- `target` which denotes the destination point of the travel.

Similarly, it would return 2 values within a tuple:

- `list` containing all nodes in the order of the quickest safe path traversal from the `source` to the `target`.
- `time` storing the total time require from reach the `target` from the `source` along the safe path.

For example, invoking the function `my_graph.quickestSafePath("0", "15")` returns the tuple `([0,2,10,16,7,4,5,15], 135.4)`.

- `[0,2,10,16,7,4,5,15]` which denotes the traversal from source 0 to target 15, going through the path of `0 -> 2 -> 10 -> 16 -> 7 -> 4 -> 5 -> 15`. Compared to the same arguments in Task 1, we are not able to traverse from vertex 10 to vertex 1 due to either vertex 1 having a red light camera; or the edge from vertex 10 to vertex 1 having a toll.
- 135.4 which denotes the total travel time (in minutes) from source 0 to target 15. Compared to the same arguments in Task 1, the safe path would be longer due to avoiding the red light cameras and toll roads.
- Similarly, note that there can be multiple quickest safe paths going from source 0 to target 15. Thus, the solution is not unique and it is fine for your program to return any of them.

If a safe path does not exist between source 0 and target 15 (i.e. vertex 15 is not reachable from vertex 0 without passing through any red light cameras or toll), then your function would return `[[],-1]` – an empty path `[]` with a traversal time of -1.

Your function should run within the time complexity of $O(E \log V)$ where V is the total number of points/ nodes/ vertices in the road network; and E is the total number of roads/ edges in the road network. The space complexity remains the same as the space complexity of your input graph $O(E + V)$.

3 Task 3: Quickest Detour Path

Often during our travels, we would need to make a detour – be it for a quick refuel, bunk for the night, grabbing a quick meal or even going on a sight-seeing. For this assignment, we would generalize this as detouring to a certain point/ location/ node/ vertex for a service. Such locations are stored within the input file `service.txt` as seen in the partial example below:

```
211
10
```

Here, we observe that vertex 211 and vertex 10 are locations in our road network that provide services to the drivers (food and gas). You are free to assume that the vertices listed in the file are all valid vertices. The rest of the unmentioned vertices are without any service.

Once again, it is up to your ingenuity on how you would represent the additional service information within your **Graph** data structure; while ensuring the complexity to still be within $O(V + E)$.

You would accomplish this using the function named `addService(self, filename_service)` within your **Graph** class. It would accept only a single argument `filename_service` which denotes the filename of the text file containing the list of service locations.

For our assignment, we define a **detour path** as the following – A **detour path** is a path from the source vertex to the target vertex; **passing through at least one** of the service vertices.

Thus, we would want to extend our quickest path solution from Task 1 (again) to now account for the additional constraint of a detour path. You would need to create a function named `quickestDetourPath(self, source, target)` within your **Graph** class. It would accept the following 2 arguments:

- `source` which denotes the starting point of the travel.
- `target` which denotes the destination point of the travel.

Similarly, it would return 2 values within a tuple:

- `list` containing all nodes in the order of the quickest detour path traversal from the `source` to the `target`.
- `time` storing the total time require from reach the `target` from the `source` along the quickest detour path.

For example, invoking the function `my_graph.quickestDetourPath("0", "15")` returns the tuple `([0,2,10,16,7,4,5,15], 135.4)`.

- `[0,2,10,16,7,4,5,15]` which denotes the traversal from source 0 to target 15, going through the path of `0 -> 2 -> 10 -> 16 -> 7 -> 4 -> 5 -> 15`. Compared to the same arguments in Task 1, we are to make a detour to vertex 16 which is our service point/ vertex before traveling all the way to the destination at vertex 15.
- 135.4 which denotes the total travel time (in minutes) from source 0 to target 15, passing through at least one service point at vertex 16. Compared to the same arguments in Task 1, the detour path would be longer due to the need to make a detour to a service node.
- Similarly, note that there can be multiple quickest detour paths going from source 0 to target 15. Thus, the solution is not unique and it is fine for your program to return any of them.

It is highly possible for the quickest path to be the same as the quickest detour path if one of the location/ point/ node/ vertex along the quickest path is already a service point. Thus, no detour is required. Your function should automatically handle such as scenario without any modification required.

If a detour path does not exist between source 0 and target 15 (i.e. vertex 15 is not reachable from vertex 0 going through at least a service point), then your function would return `[[],-1]` – an empty path `[]` with a traversal time of -1.

Your function should run within the time complexity of $O(E \log V)$ where V is the total number of points/ nodes/ vertices in the road network; and E is the total number of roads/ edges in the road network. The space complexity remains the same as the space complexity of your input graph $O(E + V)$.

Some hints (but not limited to only these hints) to help you get started on this to be within the time complexity:

- You may need to run a shortest/ quickest path traversal algorithm more than once and store the suitable values (think dynamic programming); or
- You may need to generate an additional graph with suitable edges to aid your traversal.

4 Sample Output

A sample of your program are as follows, run through the `if __name__ == "__main__":` of your Python program in the file `roadPath.py`. **All user input/ output such as user prompts, prints etc. should only be handled within the main function of your Python program.** Our tester would be calling the functions and passing arguments without any user input/ output within them.

```
---------------------------------------------------------------------
Enter the file name for the graph : basicGraph.txt
Enter the file name for camera nodes:  camera.txt
Enter the file name for the toll roads : toll.txt
Enter the file name for the service nodes : service.txt
---------------------------------------------------------------------
Source node: 0
Sink node: 15
---------------------------------------------------------------------
Quickest path:
0 --> 2 --> 10 --> 1 --> 3 --> 4 --> 5 --> 15
Time: 120.0 minute(s)
---------------------------------------------------------------------
Safe quickest path:
0 --> 2 --> 10 --> 16 --> 7 --> 4 --> 5 --> 15
Time: 135.4 minute(s)
---------------------------------------------------------------------
Quickest detour path:
0 --> 2 --> 10 --> 16 --> 7 --> 4 --> 5 --> 15
Time: 135.4 minute(s)
---------------------------------------------------------------------
Program end
```

In case of empty path, the output will be (in this example, quickest detour path is empty)

```
---------------------------------------------------------------------
Quickest detour path:
No path exists
Time: 0 minute(s)
---------------------------------------------------------------------
Program end
```

Note that none of the input files are empty.

5 Docstrings and Comments

You should use following docstring format for ALL FUNCTIONS in your program:

```
def function name([parameters]):
'''
Functionality of the function
Time complexity:
Space complexity:
Error handle:
Return:
Parameter:
Precondition:
'''
```

You can add line comments where required.

6 Crash Prevention

It is the part of the assignment to make your program safe from any kind of crash/ interruption. It is good to use all possible kind of exception handlers to save your program from being crashed/ interrupted. If your program crashed/ interrupted, the examiner/ assignment marker is not obliged to resolve the issue and/ or to mark the assignment. In this case, you will receive **zero**.

7 Use of Built-In Functions and Data Structures

You must implement the assignment 4 by using the combination of array and list as discussed in Tutorial 9 Problem 1; **NOT dictionary** or any other python built-in data structures. You must be aware of the time and space complexity of the python built-in function, if you use any python built-in functions.

-=o0o=-
END
-=o0o=-