FIT2004 S1/2019: Assignment 4

Task 1: Finding the quickest path between source and target

The solution used in Task 1 was as follows, a Graph class was initialized as per the Assignment spec requirement. Two functions were created inside the Graph class, `buildGraph(self, FileName):` and `quickestPath(self, source, target):`. Firstly, buildGraph searched through all connected vertexes in the graph in O(N) time to find the largest value, which it saved as self.maxVertex. Following this, it generates an array in O(N) time representing the all indexes up to self.maxVertex (as per the Assignment spec). This is then following by splitting each line O(1) time complexity, forming a tuple O(1) time complexity, and appending it at the index in the array we previously created O(1) time. Considering it does this for every line, it takes O(N) time, and considering that this function runs each of these loops one after the other, as opposed to running them inside one and other, and none of these loops depend on one and other, the overall time complexity is O(N) and the space complexity is O(N). The next function, quickestPath, sets all values in the array to infinity, except for our starting node, which is set to 0, as per Dijkstra's algorithm. Following this, it pushes these values onto the heap, which thereafter, the minimum value is popped, this value's neighbours then have their distances changed using the parent's distance values and their distance to the parent (parent has distance value from source, adding these values gives distance of source to child). Thus, these values are then pushed onto the heap, and the minimum value is popped, repeating the process until target is found. This results in Time complexity O(E log V) where V is the total number of nodes and E is the number of edges and Space complexity O(E + V) where V is the total number of nodes and E is the number of edges. The functions can be seen below:

```
def buildGraph(self, FileName):
    '''
    Description:        This function creates a Graph using
FileName.
    Time Complexity:    O(N).
    Space Complexity:   O(N).
    Error Handle:       Not required, is handled by the caller
function.
    Return:             Does not return a value.
    Precondition:       A FileName must be passed to this
function.
    '''
```

```
def quickestPath(self, source, target):
    '''
    Description:        This function is responsible for finding
the quickest safe path from source to target, this
                        is done by using a Dijkstra's algorithm
and a min heap (see comments for further details).
    Time Complexity:    O(E log V) where V is the total number of
nodes and E is the number of edges.
    Space Complexity:   O(E + V) where V is the total number of
nodes and E is the number of edges.
    Error Handle:       Not required, is handled by the caller
function.
    Return:             Solutions which contains the path and
minimum distance, or will return [[], -1] as per
                        Assignment spec.
    Precondition:       Source and target must be passed to this
function.
    '''
```

Task 2: Finding the quickest safe path between source and target

The solution used in Task 2 was as follows, inside our Graph class initialized in Task 1, two new functions were created. `augmentGraph(self, filename_camera, filename_toll)` and `quickestSafePath(self, source, target)`. The augmentGraph function simply reads in the file containing cameras and the file containing tolls, appending each to self.cameras and self.tolls respectively. This takes O(N) time complexity and O(N) space complexity. The next function, quickestSafePath, simply splits the self.tolls into self.tolls_left, which consists of starting point of toll, and into self.tolls_right, which consists of ending points of tolls. Every time a node is reached, it checks whether it exists in self.tolls_right, if it does, if it's parent is also in self.tolls_left and the pair exists in self.tolls, it will ignore them considering it as an invalid route. This is then done repeatedly with the valid routes, resulting in the minimum being found, similar to Task 1 (Task 2 is just built on Task 1's implementation of the heap). Thus, the time complexity of Task 2 is O(E log V), where V is the total number of nodes and E is the number of edges and the space complexity of Task 2 is O(E + V), where V is the total number of nodes and E is the number of edges. Both function descriptions can be found below:

```
def augmentGraph(self, filename_camera, filename_toll):
    '''
    Description:        This function is responsible for reading
in the cameras and tolls and storing them.
    Time Complexity:    O(N).
    Space Complexity:   O(N).
    Error Handle:       Not required, is handled by the caller
function.
    Return:             Nothing.
    Precondition:       filename_camera and filename_toll must be
passed to this function.
    '''
def quickestSafePath(self, source, target):
    '''
    Description:        This function is responsible for finding
the quickest safe path from source to target.
    Time Complexity:    O(E log V), where V is the total number
of nodes and E is the number of edges.
    Space Complexity:   O(E + V), where V is the total number of
nodes and E is the number of edges.
    Error Handle:       Not required, is handled by the caller
function.
    Return:             Returns either solutions which contains
the path and minimum distance or it will return
                        [[], -1] as per Assignment spec.
    Precondition:       Source and target must be passed to this
function.
    '''
```

Task 3: Finding the quickest detour path between source and target

The solution used in Task 3 was as follows, inside our Graph class initialized in Task 1, two new functions were created. `addService(self, filename_service):` and `quickestDetourPath(self, source, target):`. The first function, addService, simply reads in the file containing services, appending them to self.services array. This has a time complexity of O(N) and a space complexity of O(N). Building on this, the quickestDetourPath, simply iterates through each of the services in self.services, checking the distance between the source and the service, and the service and the source, if a possible path is found, it's distance is compared to either the stored variable, which starts at some arbitrarily high number, or the previous stored min if one exists, replacing it if it is of a lower distance than the one stored. This is repeated for all services, resulting in the minimum distance route being displayed. This function has a time complexity of O(E log V) where V is the total number of nodes and E is the total number of edges and a space complexity of O(E + V) where V is the total number of nodes and E is the total number of edges. A description of both functions can be found below:

```
def addService(self, filename_service):
    '''
    Description:        This function is responsible for adding
services for Task 3.
    Time Complexity:    O(N).
    Space Complexity:   O(N).
    Error Handle:       Not required, is handled by the caller
function.
    Return:             Nothing.
    Precondition:       A filename_service must be passed to this
function.
    '''
def quickestDetourPath(self, source, target):
    '''
    Description:        This function is responsible for finding
the quickest detour path between source and target.
    Time Complexity:    O(E log V) where V is the total number of
nodes and E is the total number of edges.
    Space Complexity:   O(E + V) where V is the total number of
nodes and E is the total number of edges.
    Error Handle:       If an error occurs, it signifies that a
path does not exist and is passed.
    Return:             checker which contains the route and the
minimum distance, or will return [[], -1] if no
                        route exists, as per the Assignment spec.
    Precondition:       A source and target must be passed onto
this function.
    '''
```