ID: 28390121

Name: Priscilla Tham Ai Ching

FIT2004 Assignment 4: Analysis

```
class AdjNode:

    def __init__(self, item=None, link=None):
        self.item = item
        self.next = link

class LinkedList:

    def __init__(self):
        self.head = None
        self.count = 0

    def __len__(self):
        '''
        function counts the number of items in the list
        precondition:
        :param:
        postcondition:
        :return: number of items in the list
        complexity: time: best and worst: O(1)
        '''
        return self.count

    def _get_node(self, index):
        '''
        function gets the address of the node at position index of the list
        precondition: linked list is not empty
        :param: index: position input
        postcondition:
        :return: address of the node
        complexity: time: best and worst: O(N) where N is the length of list
        error handling:
        '''
        assert 0 <= index < len(self), 'Index out of bounds.'
        node = self.head
        # when index is -1, loop runs n times
        for i in range(index):
            node = node.next
        return node

    def append(self, item):
        '''
        function appends item into array
        precondition:
        :param: item: the item to be appended into the list
        postcondition:
        :return:
        complexity: time: best: O(N) where N = 1 is possible
                          worst: O(N) where N is the length of list
        '''
        if self.head is not None:
            # when linked list is not empty, _get_index loop runs n times
            node = self._get_node(len(self) - 1)
            node.next = AdjNode(item, node.next)
        else:
            self.head = AdjNode(item, None)
        self.count += 1


class Edge:

    def __init__(self, fromVertex, toVertex, weight, toll=False):
        '''
        function creates an object of type Edge
        precondition:
        :param: fromVertex: the current vertex (current location in the road network)
                toVertex: the final vertex (destination in the road network)
                weight: the weight of the edge (road) connecting the above two vertices (travel time
between the two
                locations in the road network)
                toll: a boolean indicating the existence of toll on the road along the two locations
        postcondition:
        '''
        self.location = fromVertex
        self.destination = toVertex
        self.weight = weight
```

```
        self.toll = toll


class Vertex:

    def __init__(self, id, visited=False, status=False, camera=False, service=False):
        '''
        function creates an object of type Vertex
        precondition:
        :param: id: vertex (location) identifier
                visited: a boolean indicating if the vertex (location) is visited
                status: a boolean indicating if the vertex (location) has been finalized (left the
location)
                camera: a boolean indicating the existence of camera at the location
                service: a boolean indicating if the vertex (location) is a service point
        postcondition: each vertex will have a linked list storing the adjacent vertex and its
corresponding weight
                       and a linked list storing the adjacent edges
        '''
        self.id = id
        self.discovered = visited
        self.finalized = status
        self.camera = camera
        self.service = service
        self.adjacentVertices = LinkedList()
        self.adjacentEdges = LinkedList()

    def addAdjacent(self, vertex, weight):
        '''
        function appends adjacent vertices and edges to corresponding linked list of the vertex
        precondition:
        :param: vertex: the adjacent vertex
                weight: the weight of the edge connecting the vertex and the adjacent vertex
        postcondition:
        :return:
        complexity: time: best and worst = O(V) where V is the total number of vertices
                    space: O(V) where V is the total number of vertices
        '''
        # when list is not empty, _get_index in append runs V times
        self.adjacentVertices.append((vertex, weight))
        edge = Edge(self, vertex, weight)
        # when list is not empty, _get_index in append runs V times
        self.adjacentEdges.append(edge)
```

Above are the classes used to create the Graph. Since the required time complexity of reading file and forming the adjacency list is not specified in the assignment specification, we would skip on further elaboration.

```
class Graph:

    def __init__(self):
        '''
        function creates an adjacency list of graph representation
        precondition:
        :param:
        postcondition:
        '''
        self.graph = []
        self.detourGraph = []
        self.count = 0

    def __len__(self):
        '''
        function counts the number of vertices in the graph
        precondition:
        :param:
        postcondition:
        :return: number of vertices in the graph
        complexity: time: best and worst: O(1)
        error handling:
        '''
        return self.count

    def _removeDuplicates(self, data, data_size, column, list):
        '''
        function removes duplicates in a table and stores the filtered values in a list
        precondition:
        :param: data: the table containing duplicate values
                data_size: the number of items in the table
                column: table column
                list: the list for filtering the values
        postcondition:
        :return: filtered list
        complexity: time: best and worst: O(E) where E is the total number of edges
        error handling:
        '''
        # when number of edges == 1, loop runs 1 time
        # when number of edges > 1, loop runs V(V-1) times
        for j in range(data_size):
            vertex = data[j][column]
            if list[vertex] is None:
                list[vertex] = vertex

    def _open_file(self, filename):
        '''
        function opens a file in the same directory
        precondition:
        :param: filename: the name of the file containing the data of each vertices
        postcondition:
        :return: strings in file
        complexity: time: best and worst: O(1)
        error handling:
        '''
        with open(filename, 'r', encoding='UTF-8') as file:
            return file.read()

    def buildGraph(self, filename_roads):
        '''
        function fills in the adjacency list
        precondition: file containing the data of each vertices is not empty and data format is u v w in
each line
                      where u is a vertex and v is the adjacent vertex and w is the weight of the edge
connecting them
        :param: filename_roads: the name of the file containing the data of each vertices
        postcondition: file is not modified
        :return:
        complexity: time: best and worst: O(VE) where V is the total number of vertices and
                          E is the total number of edges
        error handling: exit function if file is not found
        '''
        try:
            # when number of edges == 1, system look for whitespace(s) once
            # when number of edges > 1, system look for whitespace(s) V(V-1) times
            content = self._open_file(filename_roads).split('\n')
            connections = len(content)
            data = []
            max = 0

            # when number of edges == 1, loop runs 1 time
            # when number of edges > 1, loop runs V(V-1) times
            for i in range(connections):
                # system looks for whitespace(s) in constant time
                values = content[i].split()
```

```
            data += [[int(values[0]), int(values[1]), float(values[2])]]
            if max < data[i][0]:
                max = data[i][0]
            if max < data[i][1]:
                max = data[i][1]

        vertices = [None]*(max+1)
        self._removeDuplicates(data, connections, 0, vertices)
        self._removeDuplicates(data, connections, 1, vertices)

        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        for j in range(max+1):
            vertex = Vertex(vertices[j])
            self.graph += [vertex]
            newVertex = Vertex(vertices[j])
            self.detourGraph += [newVertex]
            self.count += 1

        # when number of edges == 1, loop runs 1 time
        # when number of edges > 1, loop runs V(V-1) times
        # loop in addAdjacent runs (V-1) times
        # O(VE)
        for k in range(connections):
            self.graph[data[k][0]].addAdjacent(self.graph[data[k][1]], data[k][2])
            self.detourGraph[data[k][1]].addAdjacent(self.detourGraph[data[k][0]], data[k][2])
    except FileNotFoundError:
        return
```

Rather please keep in mind that the worst space complexity from building the graph which will be **used for all the tasks** is O(V+E) 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. An array of V vertices having a linked list of V-1 (assuming this is a basic graph; no loops and multiple edges to one vertex) adjacent vertices/adjacent edges totalling to V(V-1) = V^2 = E space of linked list.

TASK 1

```
def quickestPath(self, source, target):
    '''
    function finds the path with the least travel time from a location to a destination
    precondition:
    :param: source: location
            target: destination
    postcondition: adjacency list representing the graph (self.graph) is not modified
    :return: 2 values within a tuple a) list containing all locations in the order of the quickest path
traversal
            from the source to the target and b) total time required to reach the source from the target
    complexity: time: best: O(V) where V is the total number of vertices
                      worst: O(E log V) where V is the total number of vertices and E is the total number
of edges
    error handling: does not run through if graph is non-existent
    '''
    if self.count != 0:
        discoveredHeap = [None]
        finalized = []
        lookup = [-1]*self.count

        target = int(target)
        source = int(source)
        self.graph[source].discovered = True
        discoveredHeap += [[self.graph[source], 0]]

        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        while len(discoveredHeap) != 1:
            current = discoveredHeap[1]
            # system looks for item in constant time
            discoveredHeap.remove(current)
            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs (log V) times
            self._sink(discoveredHeap)

            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs V times
            for i in range(1, len(discoveredHeap)):
                lookup[discoveredHeap[i][0].id] = i

            neighbours = current[0].adjacentVertices
            if len(neighbours) != 0:
                node = neighbours.head
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs (V-1) times
                # best case when source is the target
                while node is not None:
                    weight = node.item[1] + current[1]
                    if not node.item[0].discovered and not node.item[0].finalized:
                        node.item[0].discovered = True
                        discoveredHeap += [[node.item[0], weight, current[0]]]
                        # when number of vertices == 1, loop runs 1 time
                        # when number of vertices > 1, loop runs (log V) times
                        self._rise(discoveredHeap)
                    elif not node.item[0].finalized:
                        index = lookup[node.item[0].id]
                        if index != -1 and discoveredHeap[index][1] > weight:
                            discoveredHeap[index][1] = weight
                            discoveredHeap[index][2] = current[0]
                            # when number of vertices == 1, loop runs 1 time
                            # when number of vertices > 1, loop runs (log V) times
                            self._rise(discoveredHeap)
                    node = node.next
                current[0].finalized = True
                finalized += [current]
                lookup[current[0].id] = -2

            if current[0] == self.graph[target]:
                if finalized[-1][0].id != target:
                    finalized += [current]
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs V times
                return self.calculatePath(self.graph, finalized, source, target)
    return [[], -1]
```

Task 1 requires a function *quickestPath* which will take 2 arguments: a) source which denotes the starting point of the travel and b) target which denotes the destination point of the travel.

This function implements Dijkstra algorithm to find out the quickest path from a given starting location u to a destination v. These locations are vertices within the constructed Graph class with respect to the road network in the input file. However, the file itself should not be modified.

The first *while* loops the number of vertices stored in the discovered heap – which indicates that a (next) vertex has been found – times. So, consider every vertex to be adjacent to one vertex in the graph, the number of vertices in the list will be V where V is the total number of points/nodes/vertices in the road network. A *while* loop within runs through the number of adjacent vertices of the current vertex. So, consider each vertex to be adjacent to all vertices in the graph, hence, V-1 outgoing edges (assuming this is a basic graph; no loops and multiple edges to one vertex). This loop will then run V(V-1) times where $E = V^2$ if a graph is dense.

The Dijkstra algorithm is universal, but it is modified to find the quickest path to one target. It may break the loop early once the target is found. There are such cases to consider:-

- Current vertex has no adjacent vertices
- Current vertex has adjacent vertices

Should the current vertex have no adjacent vertices, these cases apply:-

- Current vertex is the target
- Current vertex is not the target

Path will be retrieved from the finalised list of vertices – which indicates the visited vertices – when the target is found, otherwise the loop will continue. If by the end of adjacent vertices from the source and the vertices adjacent to the source the target has yet to be found, the loop ends and the function returns [[], -1].

```
def calculatePath(self, graph, list, source, target):
    '''
    function retrieves vertices in the order of the quickest path from the source to the target and
    the time taken to reach the target from the source
    precondition: list is not empty and list contains source and target
    :param: graph: adjacency list
            list: list of visited vertices (locations left)
            source: positive integer indicating the vertex id of the source
            target: positive integer indicating the vertex id of the target
    postcondition: adjacency list representing the graph (self.graph) is not modified
    :return:
    complexity: time: best and worst: O(V) where V is the total number of vertices
    error handling:
    '''
    path = []
    time = -1
    current = graph[target]
    i = len(list) - 1
    # when number of vertices == 1, loop runs 1 time
    # when number of vertices > 1, loop runs V times
    while i >= 0:
        if list[i][0] == current:
            path += [current.id]
            if current.id == target:
                time = list[i][1]
            if current != graph[source]:
                current = list[i][2]
                i = len(list) - 1
        i -= 1
    # list slicing cost V
    result = (path[::-1], time)
    return result
```

Path retrieval is a function containing a *for* looping through the finalised array which may contain up to V vertices should all vertices is adjacent to each other from the source to the target.

The current vertex is retrieved from the first position of the discovered heap which is a min-heap. This requires the usage of in-built function *remove()*. The function looks for the item in the heap array from the start. Since the item is in the first position, the operation cost constant time complexity represented in big O notation as O(1).

```
def _sink(self, heapList):
    '''
    function moves item down the heap accordingly
    precondition:
    :param: heapList: the heap array
    postcondition: heap array is stable
    :return:
    complexity: time: best and worst: O(log V) where V is the size of the heap
    error handling:
    '''
    root = 1
    # loop runs log V time(s)
    while 2 * root <= len(heapList) - 1:
        child = 2 * root
        if child + 1 < len(heapList) and heapList[child + 1][1] < heapList[child][1]:
            child += 1
        if heapList[root][1] > heapList[child][1]:
            heapList[root], heapList[child] = heapList[child], heapList[root]
        else:
            return
        root = child
```

Rearrangement of the heap is necessary after getting the minimum distanced (next) vertex to keep its nature. This uses the function *sink*. In the function sink, the *while* loop compares item in the heap at a certain position(parent node) to another item at position*2 and/or position*2+1(left and right child respectively). After rearranging, the position of each vertices in the heap is stored in a lookup array for easy access. This would need a loop through the heap to get the position and the vertex.

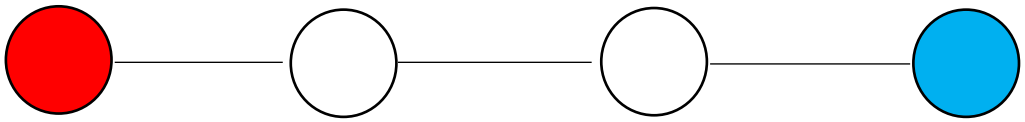
```
def _rise(self, heapList):
    '''
    function moves item up the heap accordingly
    precondition:
    :param: heapList: the heap array
    postcondition: heap array is stable
    :return:
    complexity: time: best and worst: O(log V) where V is the size of the heap
    error handling:
    '''
    n = len(heapList)-1
    # loop runs log V time(s)
    while n > 1 and heapList[n // 2][1] > heapList[n][1]:
        heapList[n // 2], heapList[n] = heapList[n], heapList[n // 2]
        n //= 2
```

When looping through the adjacent vertices, the distance to the corresponding vertex from the source vertex may be appended into/updated in the heap which again requires rearrangement. This uses the function *rise*. In the function rise, the *while* loop compares item in the heap at a certain position(left/right child) to another item at position//2 (parent node)(integer division). This is done inside the O(E) loop where E is the total number of roads/edges in the road network.

That said, we include the complexity discussed earlier on looping through the discovered heap and the adjacent vertices which is O(V) + O(E). Adding into it the time for rearrangement after taking out the next vertex O(V) * O(log V) and the time for

rearrangement after appending/updating to the heap O(E) * O(log V). The total time complexity so far is O(V log V + E log V) and we know that O(V log V) < O(E log V), hence, O(E log V). It is unnecessary to add in the complexity of updating the lookup array and the complexity of path retrieval which is O(2V) done V times, hence, O(V) * O(2V) = $O(V^2)$ < O(E) ignoring constants.

Space taken by the discovered heap, finalized array, lookup array and path array are at most O(V) in the case of all vertices is adjacent to each other from the source to the target and each vertex only has one edge. O(V) < O(V+E) which is the space taken by the graph represented by adjacency list; an array of V vertices having a linked list of V-1 adjacent vertices totalling to V(V-1) = V^2 = E space of linked list.



Red: source, blue: target

Therefore, the worst time complexity **of the function** is O(E log V) and worst space complexity **of the function** is O(V).

TASK 2

```
def quickestSafePath(self, source, target):
    '''
    function finds the safest path with the least travel time from a location to a destination
    precondition:
    :param: source: location
            target: destination
    postcondition: adjacency list representing the graph (self.graph) is not modified
    :return: 2 values within a tuple a) list containing all locations in the order of the quickest safe
path
            traversal from the source to the target and b) total time required to reach the source from the
target
            along the safe path
    complexity: time: best: O(V) where V is the total number of vertices
                      worst: O(E log V) where V is the total number of vertices and E is the total number
of edges
    error handling: does not run through if graph is non-existent
    '''
    if self.count != 0:
        discoveredHeap = [None]
        finalized = []
        lookup = [-1] * self.count

        target = int(target)
        source = int(source)
        self.graph[source].discovered = True
        discoveredHeap += [[self.graph[source], 0]]

        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        while len(discoveredHeap) != 1 and not self.graph[source].camera:
            current = discoveredHeap[1]
            # system looks for item in constant time
            discoveredHeap.remove(current)
            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs (log V) times
            self._sink(discoveredHeap)

            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs V times
            for i in range(1, len(discoveredHeap)):
                lookup[discoveredHeap[i][0].id] = i

            neighbours = current[0].adjacentEdges
            if len(neighbours) != 0:
                node = neighbours.head
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs (V-1) times
                # best case when source is the target
                while node is not None:
                    weight = node.item.weight + current[1]
                    if not node.item.destination.discovered and not node.item.destination.finalized and \
                            not node.item.destination.camera and not node.item.toll:
                        node.item.destination.discovered = True
                        discoveredHeap += [[node.item.destination, weight, current[0]]]
                        # when number of vertices == 1, loop runs 1 time
                        # when number of vertices > 1, loop runs (log V) times
                        self._rise(discoveredHeap)
                    elif not node.item.destination.finalized:
                        index = lookup[node.item.destination.id]
                        if index != -1 and discoveredHeap[index][1] > weight:
                            discoveredHeap[index][1] = weight
                            discoveredHeap[index][2] = current[0]
                            # when number of vertices == 1, loop runs 1 time
                            # when number of vertices > 1, loop runs (log V) times
                            self._rise(discoveredHeap)
                    node = node.next
                current[0].finalized = True
                finalized += [current]
                lookup[current[0].id] = -2

            if current[0] == self.graph[target] and not current[0].camera:
                if finalized[-1][0].id != target:
                    finalized += [current]
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs V times
                return self.calculatePath(self.graph, finalized, source, target)
    return [[], -1]
```

Task 2 requires a function *quickestSafePath* which will take 2 arguments: a) source which denotes the starting point of the travel and b) target which denotes the destination point of the travel. This function implements modified Dijkstra algorithm as of task 1 to find out the quickest safe path from a given starting location u to a destination v.

```
def augmentGraph(self, filename_camera, filename_toll):
    '''
    function identifies red-light camera identified vertices and toll identified edges
    precondition: file containing the list of red light cameras is not empty and format is u in
                  each line where u is a vertex and
                  file  containing the list of tolls is not empty and format is u v in each line
                  where u is a vertex and v is the adjacent vertex
    :param: filename_camera: the name of the file containing the containing the list of red light cameras
            filename_toll: the name of the file containing the list of tolls
    postcondition: file is not modified
    :return:
    complexity: time: best and worst: O(VE) where V is the total number of vertices and
                      E is the total number of edges
    error handling: exit function if file is not found/does not run through if graph is non-existent
    '''
    try:
        if self.count != 0:
            # when number of vertices == 1, system look for whitespace(s) once
            # when number of vertices > 1, system look for whitespace(s) V times
            cam_locations = self._open_file(filename_camera).split('\n')
            # when number of edges == 1, system look for whitespace(s) once
            # when number of edges > 1, system look for whitespace(s) V(V-1) times
            toll_roads = self._open_file(filename_toll).split('\n')

            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs V times
            for i in range(len(cam_locations)):
                self.graph[int(cam_locations[i])].camera = True

            toll_locations = []
            # when number of edges == 1, loop runs 1 time
            # when number of edges > 1, loop runs V(V-1) times
            for j in range(len(toll_roads)):
                # system looks for whitespace(s) in constant time
                toll_locations += [toll_roads[j].split()]

            # when number of edges == 1, loop runs 1 time
            # when number of edges > 1, loop runs V(V-1) times
            # O(VE)
            for k in range(len(toll_locations)):
                edges = self.graph[int(toll_locations[k][0])].adjacentEdges
                node = edges.head
                # when number of adjacent edges == 1, loop runs 1 time
                # when number of adjacent edges > 1, loop runs (V-1) times
                while node is not None:
                    if node.item.destination == self.graph[int(toll_locations[k][1])]:
                        node.item.toll = True
                    node = node.next
    except FileNotFoundError:
        return
```

Note though, a function *augmentGraph* which takes two filenames first containing the list of red-light cameras and second containing the list of tolls. The data of a red-light identified vertex is stored as an attribute of the vertex whereas a toll identified edge is stored as an attribute of edge. The vertices and edges are type Vertex and Edge respectively.

As of task 1, it performs the Dijkstra exactly like so with an additional condition. It will check whether the adjacent edges adjacent vertices chosen to lead towards the target has a camera by retrieving a Boolean from their attribute. It will also check whether the adjacent edges chosen to lead towards the target has a toll by retrieving a Boolean from their attribute.

The first *while* looping through the vertices in the heap may break the loop early should the source has a camera. Should the target have a camera, these cases will return [[], -1].

As discussed in task 1, the worst time complexity **of the function** is O(E log V). Space taken by the discovered heap, finalized array, lookup array and path array are at most O(V) in the

case of all vertices is adjacent to each other from the source to the target and each vertex only has one edge. O(V) < O(V+E) which is the space taken by the graph represented by adjacency list; an array of V vertices having a linked list of V-1 adjacent edges totalling to $V(V-1) = V^2 = E$ space of linked list. Therefore, the worst space complexity **of the function** is O(V).

TASK 3

```
def dijkstra(self, graph, source):
    '''
    function finds the minimum time to travel from a location to all other locations
    precondition:
    :param: graph: adjacency list
            source: location
    postcondition: adjacency list representing the graph (self.graph) is not modified
    :return: list containing all locations in the order of the quickest path
            traversal from the source to all
    complexity: time: best and worst: O(E log V) where V is the total number of vertices and
                      E is the total number of edges
    error handling:
    '''
    discoveredHeap = [None]
    finalized = []
    lookup = [-1]*self.count

    graph[source].discovered = True
    discoveredHeap += [[graph[source], 0]]

    # when number of vertices == 1, loop runs 1 time
    # when number of vertices > 1, loop runs V times
    while len(discoveredHeap) != 1:
        current = discoveredHeap[1]
        # system looks for item in constant time
        discoveredHeap.remove(current)
        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs (log V) times
        self._sink(discoveredHeap)

        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        for i in range(1, len(discoveredHeap)):
            lookup[discoveredHeap[i][0].id] = i

        neighbours = current[0].adjacentVertices
        node = neighbours.head
        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs (V-1) times
        while node is not None:
            weight = node.item[1] + current[1]
            if not node.item[0].discovered and not node.item[0].finalized:
                node.item[0].discovered = True
                discoveredHeap += [[node.item[0], weight, current[0]]]
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs (log V) times
                self._rise(discoveredHeap)
            elif not node.item[0].finalized:
                index = lookup[node.item[0].id]
                if index != -1 and discoveredHeap[index][1] > weight:
                    discoveredHeap[index][1] = weight
                    discoveredHeap[index][2] = current[0]
                    # when number of vertices == 1, loop runs 1 time
                    # when number of vertices > 1, loop runs (log V) times
                    self._rise(discoveredHeap)
            node = node.next
        current[0].finalized = True
        finalized += [current]
        lookup[current[0].id] = -2
    return finalized
```

Task 3 requires a function *quickestDetourPath* which will take 2 arguments: a) source which denotes the starting point of the travel and b) target which denotes the destination point of the travel. This function implements Dijkstra algorithm to find out the quickest path from a given starting location u to a destination v passing by a service point in between.

```
def addService(self, filename_service):
    '''
    function identifies service point identified vertices
    precondition: file containing the list of service locations is not empty and format is u in
                  each line where u is a vertex and
    :param: filename_service: the name of the file containing the list of service locations
    postcondition: file is not modified
    :return:
    complexity: time: best and worst: O(V) where V is the total number of vertices
    error handling: exit function if file is not found/does not run through if graph is non-existent
```

```
    '''
    try:
        if self.count != 0:
            # when number of vertices == 1, system look for whitespace(s) once
            # when number of vertices > 1, system look for whitespace(s) V times
            servicePoints = self._open_file(filename_service).split('\n')

            # when number of vertices == 1, loop runs 1 time
            # when number of vertices > 1, loop runs V times
            for i in range(len(servicePoints)):
                self.graph[int(servicePoints[i])].service = True
                self.detourGraph[int(servicePoints[i])].service = True
    except FileNotFoundError:
        return
```

Note though, a function *addService* which takes a filename containing the list of service points. The data of a service point identified vertex is stored as an attribute of the vertex. The vertices are type Vertex.

The Dijkstra algorithm is universal so, we would skip on further elaboration but the implementation to find out whether the path retrieved pass by a service point is done by comparing the path (A) retrieved by Dijkstra from the source and the path (B) retrieved by Dijkstra from the target. The algorithm is such that we will first retrieved the service point identified vertices throughout the B. Vertices in B are vertices reachable by the target. Should it be none, the function returns [[], -1].

```
def quickestDetourPath(self, source, target):
    '''
    function finds the path with the least travel time from a location to a destination passing through at
least
    one of the service point
    precondition:
    :param: source: location
            target: destination
    postcondition: adjacency list representing the graph (self.graph/self.detourGraph) is not modified
    :return: 2 values within a tuple a) list containing all locations in the order of the quickest path
traversal
            from the source to the target and b) total time required to reach the source from the target
along the
            quickest detour path
    complexity: time: best and worst: O(E log V) where V is the total number of vertices and
                    E is the total number of edges
    error handling: does not run through if graph is non-existent
    '''
    if self.count != 0:
        target = int(target)
        source = int(source)
        finalized = self.dijkstra(self.graph, source)
        reversedFinalized = self.dijkstra(self.detourGraph, target)

        reachable = []
        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        for j in range(len(reversedFinalized)):
            if reversedFinalized[j][0].service:
                reachable += [reversedFinalized[j]]

        finalizedPath = None
        time = float('inf')
        # when number of vertices == 1, loop runs 1 time
        # when number of vertices > 1, loop runs V times
        for k in range(len(reachable)):
            fstHalf = self.calculatePath(self.graph, finalized, source, reachable[k][0].id)
            sndHalf = self.calculatePath(self.detourGraph, reversedFinalized, target, reachable[k][0].id)
            timeTaken = fstHalf[1]+sndHalf[1]
            if len(fstHalf[0]) != 0 and time > timeTaken:
                path = []
                path += fstHalf[0]
                # when number of vertices == 1, loop runs 1 time
                # when number of vertices > 1, loop runs V times
                for m in range(len(sndHalf[0])-2, -1, -1):
                    path += [sndHalf[0][m]]
                finalizedPath = path
                time = timeTaken

        if finalizedPath is not None:
            result = (finalizedPath, time)
```

```
            return result
    return [[], -1]
```

After retrieving the service point identified vertices reachable from the target, we retrieved the path (new A) from the source to the service point and the path (new B) from the target to the service point. If A is empty, source and target does not share a path to this service point and we move on to the next reachable service point from the target. Else, the time taken on this path will then be compared to the time taken on the next path passing through the next reachable service point. The function will return the path with the minimum time taken.

The first for loops through the finalised array retrieved from the target as source Dijkstra. Again, as mentioned in the two tasks above: this may contain up to V vertices should all vertices is adjacent to each other from the source to the target and all vertices are service points.

Next for loops through the array formed from the reachable service points above. This loop runs a maximum of O(V). A path retrieval function within is the same function used in both tasks above. It contains a *for* looping through the finalised arrays which may contain up to V vertices should all vertices is adjacent to each other from the source to the target. The target in this case is the service point(s). The retrieved paths will be concatenated to form a full path.

It is unnecessary to add in the complexity of the two loops $O(V) + O(V^2)$ as $O(V) < O(V^2) < O(E) < O(E \log V)$ which is the time complexity of the Dijkstra ignoring constants. Space taken by the discovered heap, finalized array, lookup array and path array are at most O(V) in the case of all vertices is adjacent to each other from the source to the target and each vertex only has one edge and each vertex is a service point. $O(V) < O(V+E)$ which is the space taken by the graph represented by adjacency list. Therefore, the worst time complexity **of the function** is O(E log V) and worst space complexity **of the function** is O(V).