AI61005
Assignment-1
Problem 3 – Electric vehicles
ROUTE FINDING ALGORITHM

Team members:

1) Nachiketh B M, 19ME30028

Problem statement:

The assignment is to solve an electric vehicle problem by finding the best possible route to minimize the maximum time taken by the electric vehicles which are travelling across cities.

Approach:

I have approached the assignment by incorporating heuristic and optimal algorithm into two python files. Optimization is done in the 'Optimize.py' file, and heuristic is developed in the 'Route_Finding_Algorithm.py' file. The third file namely 'execute.py' uses both of these files, takes the following inputs mentioned below from the user, executes both heuristic algorithm and optimization and returns the heuristic matrix and the total time taken by all the vehicles as output.

Details:

1) Language used: Python 100%
2) External Libraries used: NumPy only
3) External algorithms used: None
4) Python IDE used: PyCharm
5) A brief description of each file:

   (a) Route_Finding_Algorithm: Calculates distance heuristic matrix by finding the shortest route via an intermediate node. Also calculates total time taken in the journey by all vehicles before optimization.

   (b) Optimize: Minimizes the total time of that vehicle which requires the highest time by giving it the priority in the source node charging station and also in the intermediate node charging station to charge first.

   (c) Exexute: Imports the above files, takes the required inputs directly from the user, converts it into required format, applies the function with inputs as arguments and finally prints the output to the user.

Here is how the execute file performs the above-mentioned functions:

```python
import numpy as np
import Route_Finding_Algorithm

# Getting inputs

n = int(input("Enter number of cities: "))
e = np.array(input("Enter distance between cities: ").split(), float)
e = e.reshape(n, n)

k = int(input("Enter number of electric vehicles: "))

# s_node-source node   d_node-destination node   b-battery charge status initially
# c-charging rate   d-discharging rate   m-maximum battery capacity   s-average travelling speed

s_node = np.array(input("Enter source node: ").split(), int)
d_node = np.array(input("Enter destination node: ").split(), int)

initial_battery = np.array(input("Enter initial battery charge status: ").split(), float)
# unit: kW hr
charging = np.array(input("Enter charging rate: ").split(), float)
# unit: kW
discharging = np.array(input("Enter discharging rate: ").split(), float)
# unit: kW
maximum_battery = np.array(input("Enter maximum battery capacity: ").split(), float)
# unit: kW hr
speed = np.array(input("Enter average travelling speed: ").split(), float)
# unit: km/hr

# Applying Route_Finding_Algorithm

time, heuristic = Route_Finding_Algorithm.find_route(n, e, k, s_node, d_node, initial_battery, charging, discharging, maximum_battery, speed)
print(heuristic)
print(time)
```

Inputs:

```
C:\DS\Anaconda3\python.exe "C:\Users\Nachiketh BM\Desktop\Pycharm\AIFA Assignment\execute.py"
Enter number of cities: 5
Enter distance between cities: 0 60 0 45 0 60 0 29 71 0 0 29 0 0 81 45 71 0 0 42 0 0 81 42 0
Enter number of electric vehicles: 6
Enter source node: 0 0 1 2 1 4
Enter destination node: 3 4 2 4 0 3
Enter initial battery charge status: 25 40 75 100 90 85
Enter charging rate: 15 17 19 21 25
Enter discharging rate: 0.7 0.8 0.9 1.0 1.1 1.2
Enter maximum battery capacity: 100 100 100 100 100 100
Enter average travelling speed: 35 50 40 75 65 55
```

Output for the above input:

```
Heuristic =  [[  0.  60.  89.  45.  87.]
 [ 60.   0.  29.  71. 110.]
 [ 89.  29.   0. 100.  81.]
 [ 45.  71. 100.   0.  42.]
 [ 87. 110.  81.  42.   0.]]
Time =  [3.9047619  5.44238095 0.725      1.08       0.92307692 0.76363636]

Process finished with exit code 0
```

NOTE: For the distance matrix eij, please enter zero for all those edges which are not directly connected, though ‘infinity’ was recommended in the problem statement.

Algorithm content:

execute.py

Important variables:

Variable	Use	Data type
n	number of stations	Int
e	distance between the cities	2D numpy array(n*n)-float
k	number of vehicles	Int
s_node	source node	1D numpy array(k)-int
d_node	destination node	1D numpy array(k)-int
Initial_battery	Initial battery of a vehicle	1D numpy array(k)-float
charging	Charging rate at a station	1D numpy array(n)-float
discharging	Discharging rate of a vehicle	1D numpy array(k)-float
maximum_battery	Maximum battery of a vehicle	1D numpy array(k)-float
speed	Average speed of a vehicle	1D numpy array(k)-float

1) Imports Route_finding_Algorithm and NumPy.
2) Stores the user inputs in the above defined variables after converting them into a numpy array of specific dimensions and length.
3) Applies the find_route function in the Route_finding_Algorithm by passing the above variables as arguments.
4) Stores the output in heuristic and time variable and prints them to the user.

Route_finding_Algorithm.py

Important variables:

Variables	Use	Data type
node_tested	Stores all the nodes tested while calculating the heuristic	Dictionary
node_sum	Stores all the possible intermediate nodes for a given source and destination node	Dictionary
heuristic	Stores the least possible calculated distances along with original distance in a matrix	2D numpy array(n*n)-float
Intermediate node	Lists the intermediate nodes through which a vehicle passes while travelling from source node to destination node	2D list(n*n)-int

time	Stores the total time taken by all the vehicles	1D numpy array(k)-float
battery_level	Stores the battery level of all vehicle at any time	Dictionary
required_charge	Stores what charge is required by a vehicle in its complete travel	Dictionary
required_time_start	Stores what time is wasted in getting charged or waiting for it at source node	Dictionary

required_time_middle	Stores what time is wasted in getting charged or waiting for it at its intermediate node	Dictionary
begin_time_middle	Stores at what time the vehicle has reached the intermediate node	Dictionary
middle_node	Stores the intermediate node for a particular vehicle	int
Station_node	Stores all the vehicles starting from or passing through a particular node	3D List(n*2*variable)

1) The function calculate_distance takes heuristic matrix, a row index(source node) and a column index(destination node) as input. It finds the most suitable intermediate node common to both source node and destination node by iterating through different nodes of the given input column. It marks node_tested[i] as True if the node is already checked, if any node satisfies the constraints to be a intermediate node, that node is stored in node_sum dictionary. Finally the function checks the best possible intermediate node by comparing the total distances and returns both the node index and total distance between the given source and destination node.
2) The function calculate_heuristic takes total number of stations(int) and distances between those stations(2D NumPy array) as input. Another 2D NumPy array is defined inside the function which takes all the non zero values from the input distance matrix and if two cities are not connected directly then the calculate_distance function is executed for that particular row and column and the middle node is stored in a list called intermediate_node which stores the node index as a value at row=source node and column=destination node. Finally the function returns both the NumPy heuristic matrix and also the intermediate node list.

```
    i1 = 0
    while i1 < n:
        i2 = 0
        while i2 < n:
            if heuristic[i1][i2] == 0 and not i1 == i2:
                min_value, min_index = calculate_distance(heuristic, i1, i2)
                heuristic[i1][i2] = min_value
                intermediate_node[i1][i2] = min_index
            i2 = i2 + 1
        i1 = i1 + 1

    return heuristic, intermediate_node
```

3) The function find_route is the heart of the algorithm and takesall the variables defined in execute file as inputs.

```
def find_route(n, e, k, s_node, d_node, initial_battery, charging, discharging, maximum_battery, speed):
```

Almost all the variables are defined in the beginning of function. Initially the time array is set to zeros, the battery_level is given the initial_battery given directly by the input user and some empty dictionaries required_charge, required_time_start, required_time_middle, begin_time_middle are defined to incorporate the above mentioned data. We then iterate through every vehicle and we first find the required charge for the vehicle to go to the next nearest station, if the maximum battery is not enough for the travel, 'None' is returned in the output time matrix time. Whereas if the maximum battery is sufficient enough to take the vehicle to the next nearest station, the battery_level is checked and it's sufficiently charged. The time required for charging is stored in required_time_start and after the vehicle reaches the middle node(the time is stored is begin_time_middle), it would have lost some charge and it is required to charge again and this time is stored in required_time_middle. The above steps are repeated again from the middle node to the destination node and the final time is calculated according to the formula:

time(k) = heuristic(i, j) / speed(k) + starting_time + final_time - initial_time

After the total time is calculated for all the vehicles we define an empty list called station_node. Using multiple for loops, we store all the vehicles passing through a node(either source node or intermediate node, not destination node) and we store the vehicle index in station node. Finally the optimization of the time is done by applying the optimize algorithm(which was imported in the beggining) which returns the modified time after optimizing. In the last line the function return this optimized time along with the heuristic matrix coming from calculate_heuristic function.

```
time = Optimize.optimize(n, time, required_time_start, required_time_middle, begin_time_middle, station_node)

return time, heuristic
```

Optimize.py

Important variables:

Variable	Use	Data type
Station_free_at	Gives the time when the station is free after charging other vehicles	int
time_occupied	It is a large array storing Boolean variables which informs whether the station is busy in that second	1D numpy array

priority_order	Stores the priority to be given during optimization in a descending order	1D List
max_priority_vehicle	Temporarily stores the vehicle index with max priority	int
vehicle_available	Used to indicate the usage of a particular vehicle while iterating	Dictionary
t_max	The maximum time until which the algorithm considers for finding if station is busy	1 D List
t_initial	The time at which a vehicle enters the intermediate node	int
t_final	The time at which the vehicle leaves the intermediate node	int

t_initial_new	The time at which the vehicle starts getting charged in the intermediate node	int

The optimize function takes total stations, time matrix and some dictionaries such as required_time_start, required_time_middle, begin_time_middle, station_node as inputs. By iterating through all the stations, the function first checks whether any vehicle passes through that particular station as source node. If yes, the list 'priority_order' stores the priority to be given to the vehicles for charging at that particular station in a descending order. Then the time matrix is modified according the priority order by giving first preference to that vehicle which comes first in the priority order. Then the function checks whether any vehicle passes through that particular station as intermediate node. If yes, the list 'priority_order' again stores the priority to be given to the vehicles for charging at that particular station. Now we define t_max and allot it a rough number which will be the upper limit until which we check whether the station is occupied in that particular second or not. We define the NumPy array 'time_occupied' and set it True for all t from t=0 to t=t_max.

```
            # Assuming energy given per unit time at any station is not greater than 10 times
            # of the energy lost per unit time during travelling(thinking practically)
            t = 0
            t_max = 10*3600*(time[priority_order[0]])
            t_max = int(t_max - t_max % 1)
            time_occupied = np.zeros(t_max)
            while t < t_max:
                time_occupied[t] = False
                t = t + 1

            i2 = 0
            while i2 < len(row):
                t_initial = begin_time_middle[row[i2]]
                t_final = begin_time_middle[row[i2]] + required_time_middle[row[i2]]
                time_occupied, t_initial_new = check_time(time_occupied, t_initial, t_final)
                time[row[i2]] = time[row[i2]] - t_initial_new + t_initial
                i2 = i2 + 1
    i1 = i1 + 1

return time
```

Finally we iterate along the rows of station node and modify the time after checking whether the time required for the vehicle to charge is occupied or free using a function check_time. The function modifies ‘time_occupied’ if the station is free and it returns station_free_at = 0, as it is free at that particular time. If the station is not free, the function checks -until what time the station is busy and at what time it becomes free for the purpose of charging that particular vehicle and also it modifies the time_occupied list. After the iterations are complete for all the stations and time is modified for all the vehicles, the algorithm finally returns the modified ‘time’ matrix which is also optimized.

THE END