

**TITLE**: OPTIMIZATION OF SMART AND EFFICIENT NAVIGATION FOR ELECTRIC VEHICLES CONSIDERING GEOGRAPHY AND BATTERY LIFE.

**Group number : CSE-G50**

* **Supervisor Name** : Ms. Radhika Sreedharan

**GROUP MEMBERS:**

1. P. Ganesh Babu(20191CSE0438)
2. U. Kranthi Kiran(20191CSE0650)
3. Rohan L Bhattar(20201LCS0016)
4. V. Sai Puneeth(20191CSE0695)
5. V. Jitendra(20191CSE0694)

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S. No** | **Title** | **Page no.** |
| 1 | Summary of Base paper | 4-5 |
| 2 | Abstract | 5-6 |
| 3 | Literature Survey | 6 |
| 4 | Methodology | 7-8 |
| 5 | Analysing of Existing system | 9-14 |
| 6 | Source Code | 15-28 |
| 7 | Result | 29 |
| 8 | Proposing a new System | 30-35 |
| 9 | Source Code | 36-49 |
| 10 | Result | 50 |
| 11 | Conclusion | 51 |
| 12 | References | 52 |

**List of Figures**

|  |  |  |
| --- | --- | --- |
| Fig. No | Title | Page no |
| 1 | Area Network | 9 |
| 2 | Shortest Paths | 11 |
| 3 | Graph that depicts at what time the battery starts charging | 12 |
| 4 | Graph that depicts at what time the battery ends charging | 13 |
| 5 | Graph that depicts at what time EV leaves a particular node | 14 |
| 6 | Area Network | 30 |
| 7 | Shortest Paths | 32 |
| 8 | Graph that depicts at what time the battery starts charging | 33 |
| 9 | Graph that depicts at what time the battery ends charging | 34 |
| 10 | Graph that depicts at what time EV leaves a particular node | 35 |

**List of Tables**

|  |  |  |
| --- | --- | --- |
| Table No | Name of the Table | Page No |
| 1 | Base Paper Details | 4 |
| 2 | Literature Survey | 6 |
| 3 | Node to Node Distances | 10 |
| 4 | EV Details | 11 |
| 5 | Node to Node Distances | 31 |
| 6 | EV Details | 31 |

**CHAPTER 1**

**SUMMARY OF THE BASE PAPER**

* 1. **BASE PAPER DETAILS**

Table 1.1. Base Paper Details

|  |  |
| --- | --- |
| **Title** | Energy efficient route planning for electric vehicles with special consideration of the topography and battery lifetime. |
| **Authors** | Theresia Perger ,Hans Auer |
| **Journal Name** | Energy Efficiency |
| **Year of Publishing** | 2020 |
| **Publisher** | Springer |
| **ISSN Number** | 1705–1726 |

**INTRODUCTION**

Our world has been ruled by cars powered by combustion engines for more than a century. Electric Vehicles could be a part of the solution to future environmental regulations and worries about global warming, with the requirement that the electrical power originate from renewable sources. With this new transportation technology option come additional difficulties. Even if internal combustion engines are more efficient, it is still difficult to store energy in a vehicle. Comparing the battery's range to that of normal cars, it is significantly less. In addition, refuelling an electric vehicle takes longer than charging one. This makes journey planning with an electric vehicle more difficult. Knowing where the nearest charging stations are in advance will help you plan your itinerary. The approach taken will have a considerable impact on the energy use. This study elaborates on the best way to construct a route while taking electric vehicles' unique qualities into account. The major goals are to extend battery life and reduce journey time and energy consumption considering topographic effects. Finding of shortest route between a starting and an ending point while using the least amount of energy is the first goal of this study. This process is made more complex so that it is feasible to determine the fastest path and the optimal route to extend battery life. A weighted multi-objective optimization problem is created using these route planning alternatives. In networks with nodes, edges, and edge costs, shortest path techniques are employed. We can find a number of routes for any electric vehicle, as well as different places and rates for battery charge. Because it is difficult to move about the state space, various assumptions were made:

1) Only discrete charges, or integer percentages, are permitted for batteries. Because the difference in battery charging values will always fall between [0, 1], this won't significantly effect optimality.

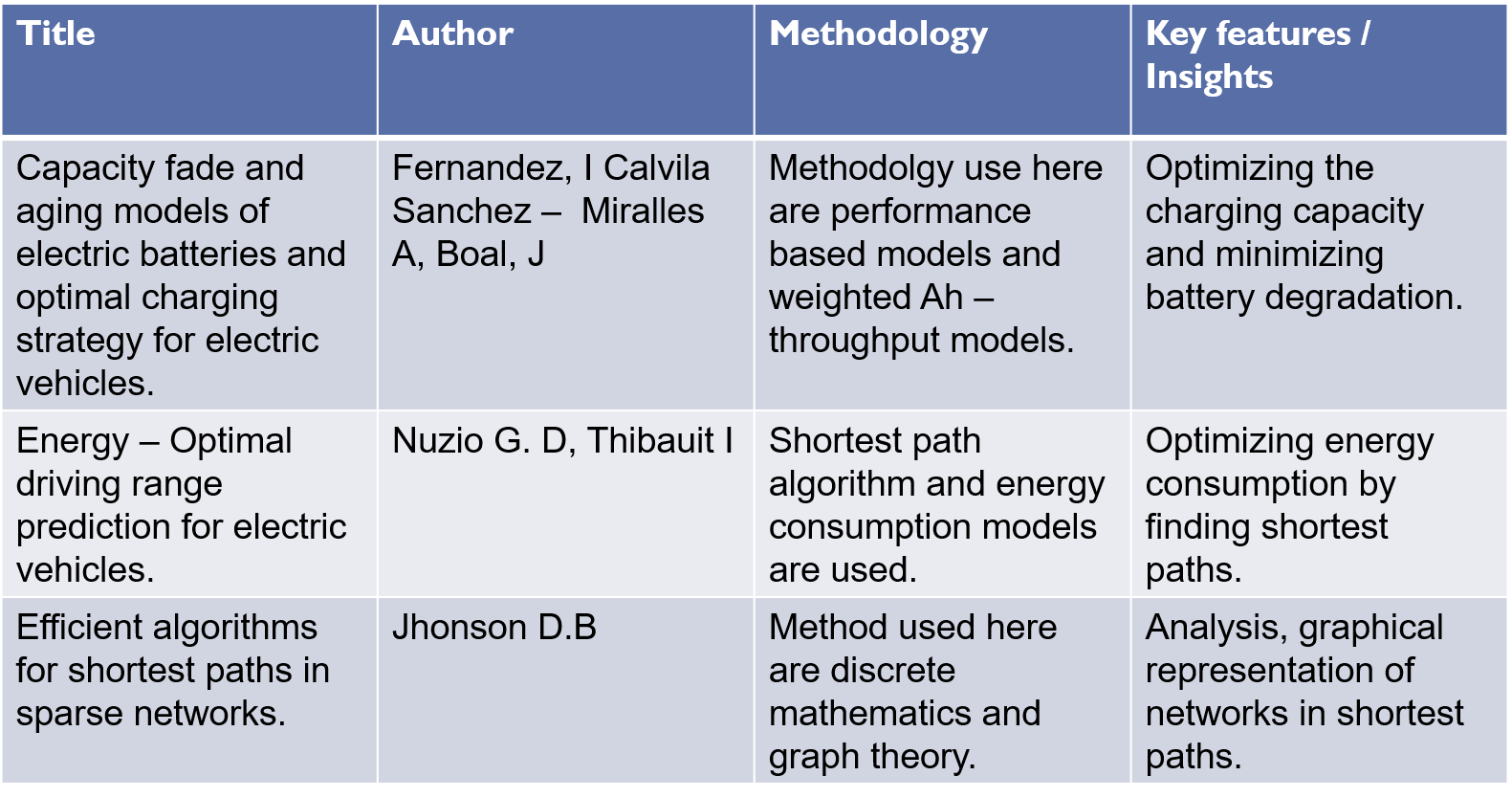
2) Maximum depth, the number of routes we will take into consideration for each electric car, has been initialised. The computation takes longer when all possible pathways are taken into account. maximum depth can therefore be initialised to any integer (2, 3, 4, etc.). Complete combinations of pathways will be taken into consideration if maximum depth is set very high.

**ABSTRACT**

In general, there are many ways to design the route of the vehicle that determines the route.

Although it is the shortest route to the destination, our project mainly is about electric vehicles by finding load-aware energy optimization solutions. In concern to this optimization, variables are assigned with weights to emphasize one or the other. We will determine the energy consumption of the Electric vehicles model. Now a network area of the considered street is modelled with nodes and weighed edges, so that a shortest path finding algorithm is implemented which finds the path with cheapest edge. The route planning system will be tested in the locality of Hyderabad. Power consumption of vehicle and battery performance mainly depends on topography of the area, also selection of route. An existing system is first studied and simulated. A system is later suggested by creating an area network with the aid of nodes. Plotting several graphs for the pace of battery charging and discharging for a set of electric vehicles follows the design of the area network for the chosen suburban region in Hyderabad. With reference to an existing system that we previously examined and simulated, the suggested system is now simulated, and results are seen.

**Literature Survey:**



**CHAPTER 2**

**METHODOLOGY**

The main goal of this project is to design a routing system for Electric Vehicles that

maximize battery life, energy consumption and run time.

**STEPS:**

1) For each electric car, create all feasible routes and their respective distances.

2) Determine the best maximum depth number of routes by sorting the routes according to their distances.

3) Produce a matrix for charging batteries. For this, the recursive function time

calculator() was employed. Numerous logical factors, such as the feasible charging

range and the overall charge required, have been taken into account.

4) After each matrix has been created, the final time() function is invoked, which

assigns the appropriate operations based on charge times, priorities, and end times.

5) The max(Time of electric vehicle) is compared with globally maintained variables.

**FLOWCHART:**

**Start**

**Set start and destination position**

**Get map of road network**

**Initialise vehicle parameters**

**Calculate energy consumption and journey time of all road sections**

**Select the shortest path**

**If batterycharge>Journey time**

**yes**

No

Yes

**Find Optimum**

**Reach Destination**

**End**

**CHAPTER 3**

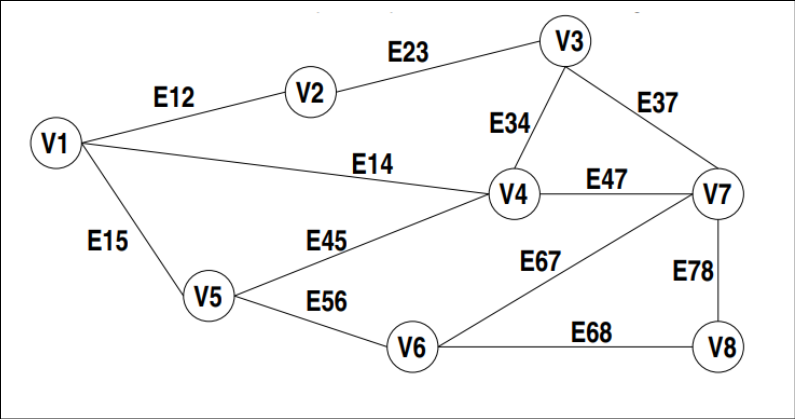
**ANALYSING OF EXISTING SYSTEM**

Consider a city network where we need to route a set of electric vehicles which may require to be charged during its journey from some source to some destination.Assume that each city has a single charging station which can charge one EV at a time. Consider a set of k EVs namely P1, P2, . . . , Pk. For each EV the following information is provided.

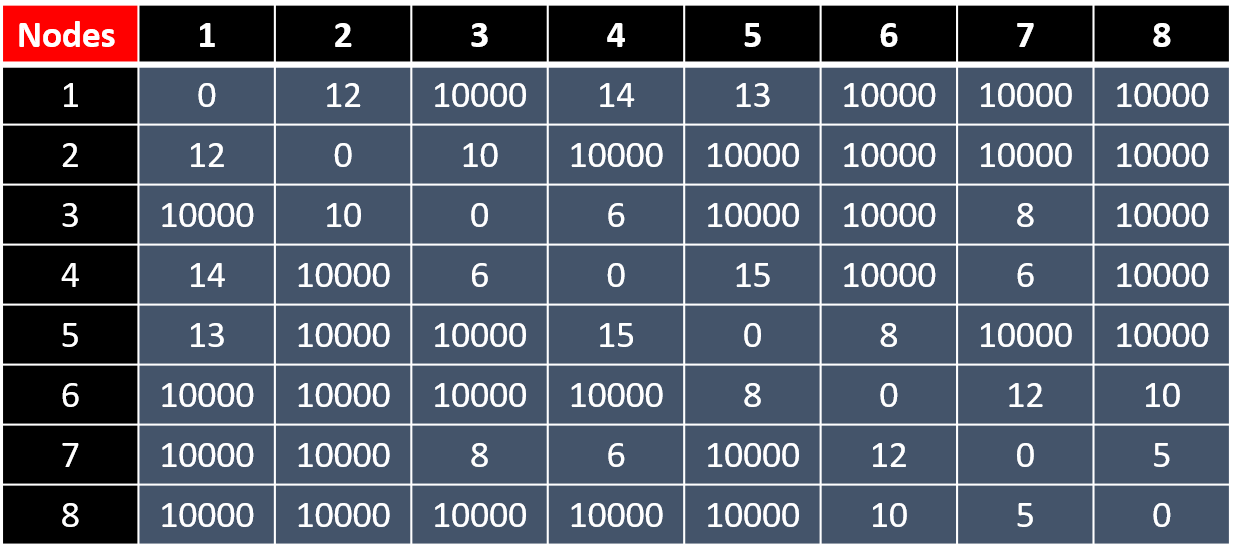
1. source node
2. destination node
3. battery charge status initially
4. charging rate for battery at a charging station (percent charged per unit time)
5. discharging rate of battery while traveling (distance travel per unit charge)
6. maximum battery capacity
7. average traveling speed (distance per unit time).

Assume that all vehicles start their journey at t = 0 and Pr reaches it destination at t = Tr. We need to route all the vehicles from their respective sources to destinations such that max{Tr} is minimized.

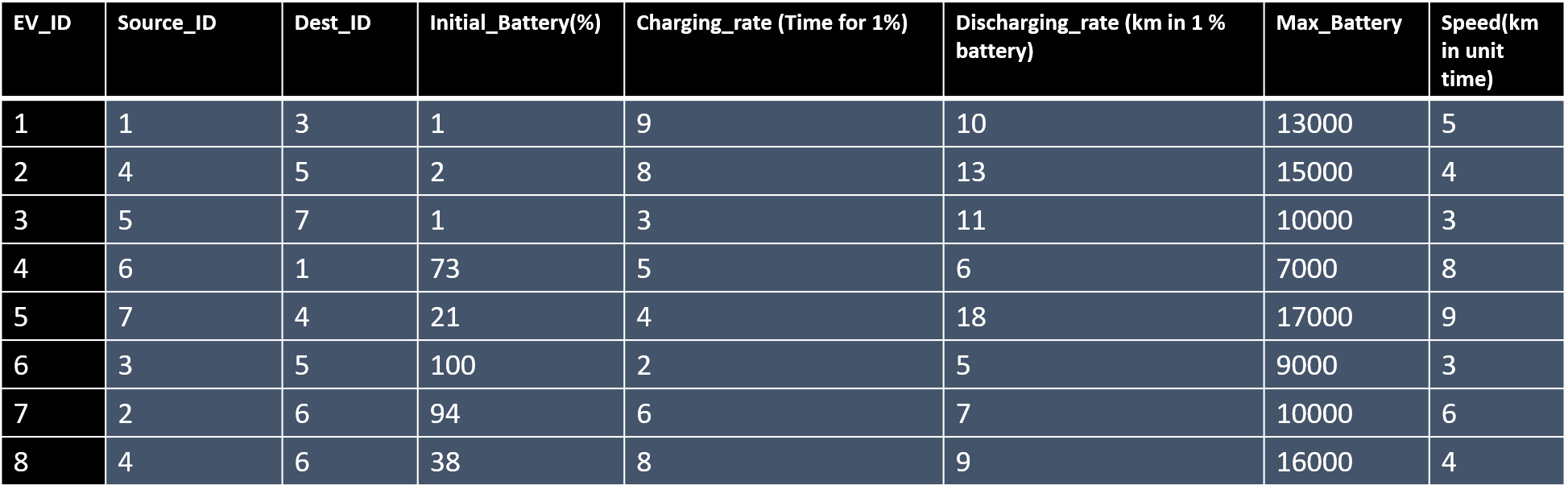
**AREA NETWORK:**



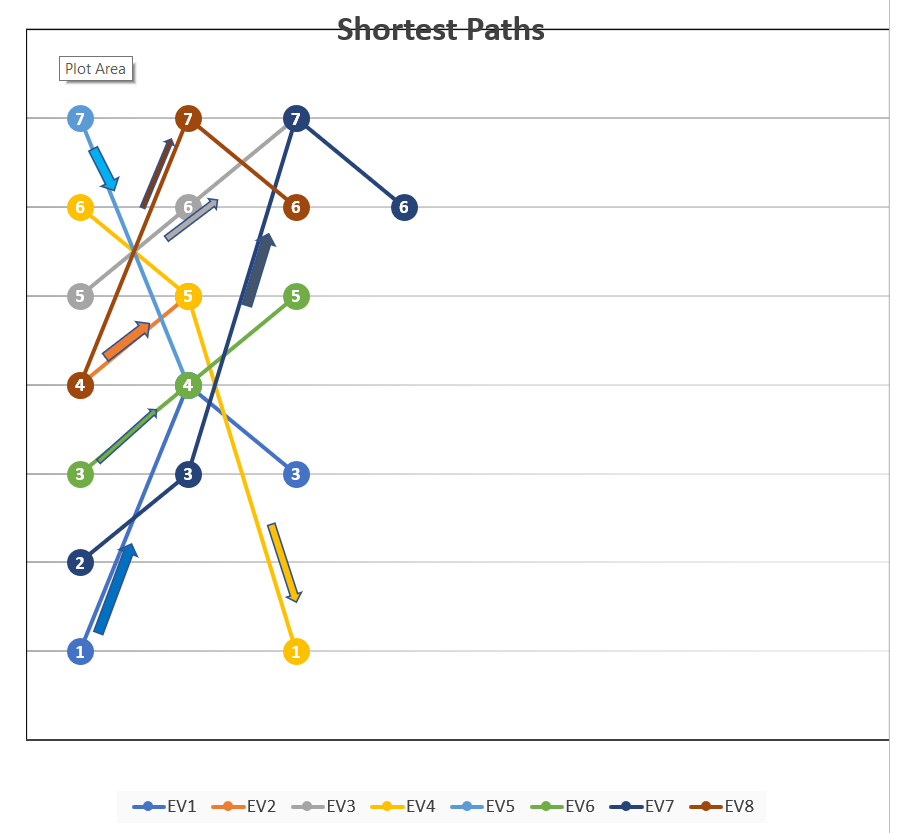
**NODE TO NODE DISTANCES:**



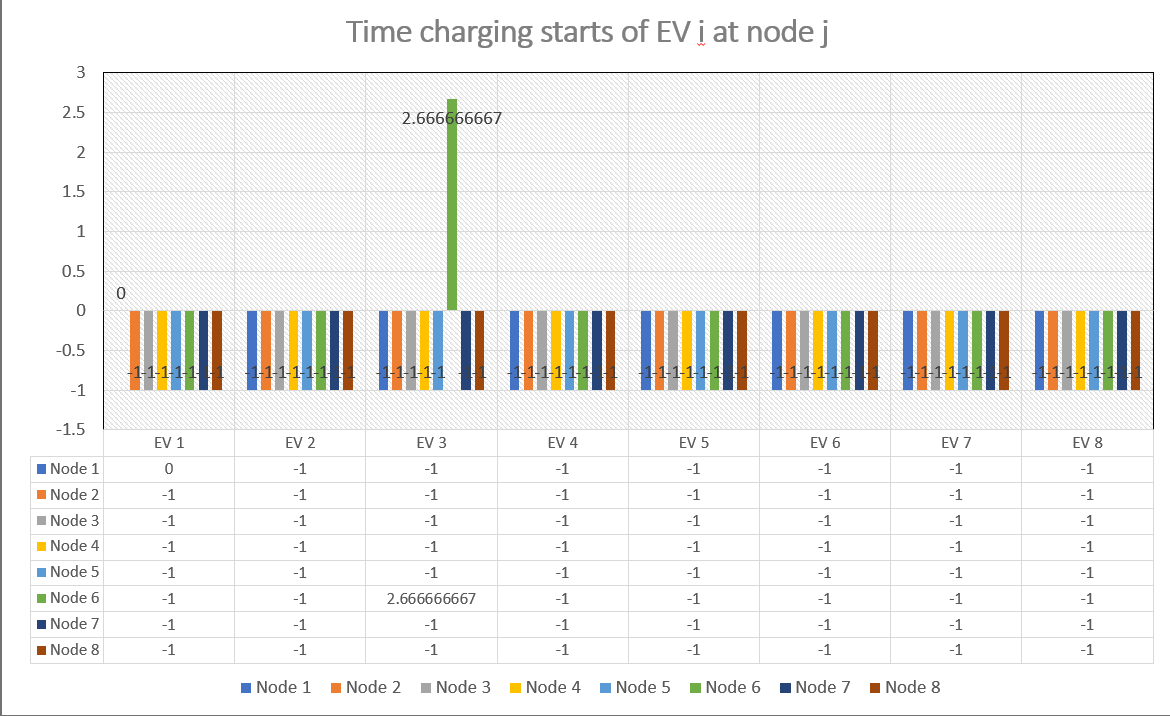
**EV DETAILS:**



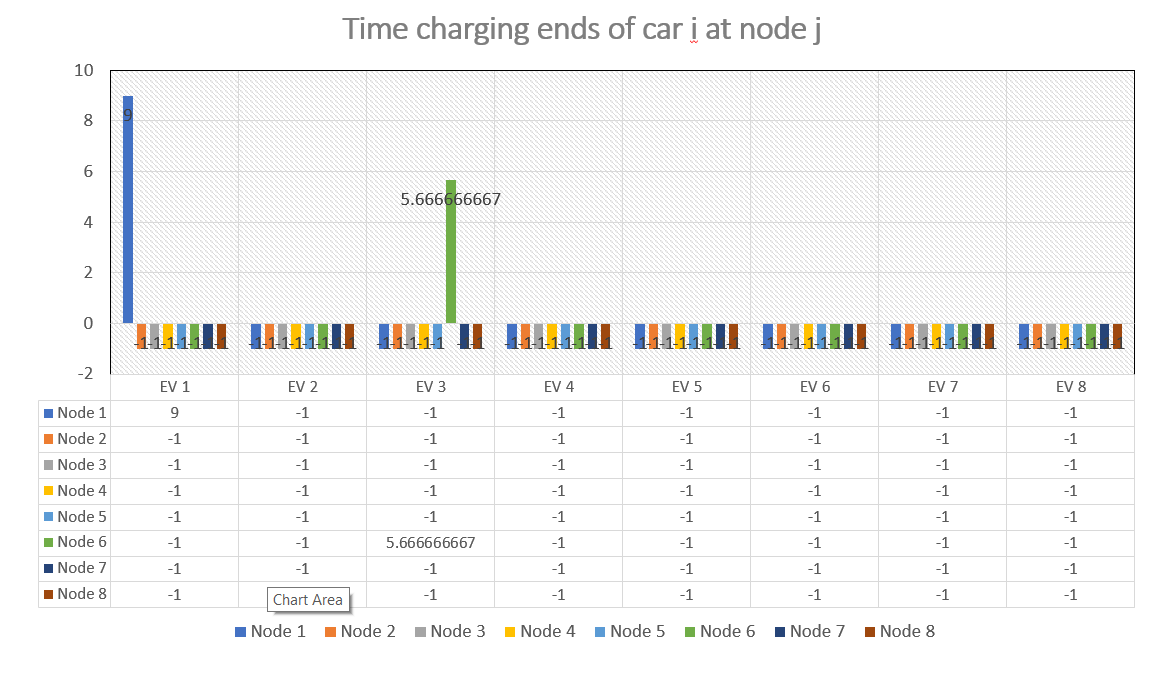
**Required paths that are to be taken by the EVs:**

****

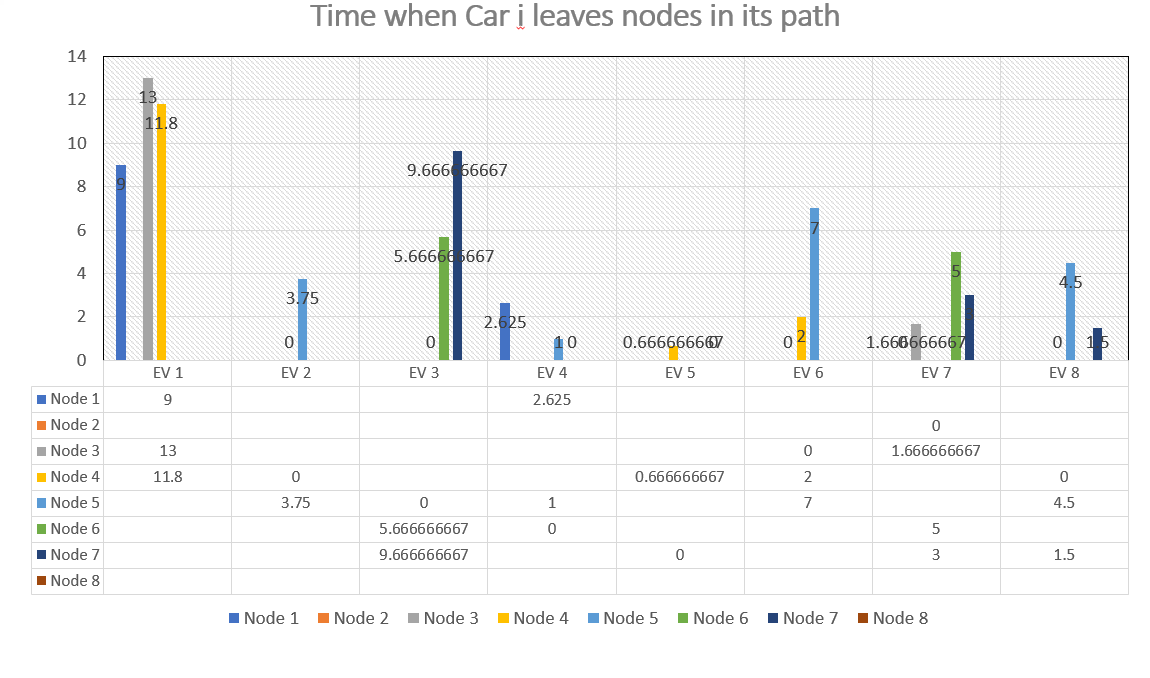
**The below graph depicts at what time the battery starts charging:**

****

**The below graph depicts at what time the battery ends charging:**

****

**The below graph depicts at what time EV leaves a particular node:**

****

* The Maximum time (Tr) can be calculated through this.
* This is the already existing system, with reference to this existing system a new system is proposed and simulated.

**SOURCE CODE OF EXISTING SYSTEM**

import sys

import numpy as np

import math

noCity = 8

noCar = 8

maxDepth = 2

class Cars():

def \_init\_(self, quantity):

self.V = quantity

self.stats = [[0 for column in range(7)]

for row in range(quantity)]

self.route\_all = [[0 for column in range(1)]

for row in range(quantity)]

self.route\_dist\_all = [[0 for column in range(1)]

for row in range(quantity)]

self.route = [[0 for column in range(1)]

for row in range(quantity)]

self.route\_dist = [[0 for column in range(1)]

for row in range(quantity)]

ev = Cars(noCar)

q = np.genfromtxt(r"C:\Users\prath\OneDrive\Desktop\EV PROJECT\EV.csv", delimiter=',')

q = q.tolist()

ev.stats = q

r = np.genfromtxt(r"C:\Users\prath\OneDrive\Desktop\EV PROJECT\Nodes.csv", delimiter=',')

r = r.tolist()

r[0][0] = 0

# Python program to print all paths from a source to destination.

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph2:

def \_init\_(self, vertices):

# No. of vertices

self.V = vertices

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self, u, v):

self.graph[u].append(v)

'''A recursive function to print all paths from 'u' to 'd'.

visited[] keeps track of vertices in current path.

path[] stores actual vertices and path\_index is current

index in path[]'''

def printAllPathsUtil(self, u, d, visited, path, id):

# Mark the current node as visited and store in path

visited[u]= True

path.append(u)

# If current vertex is same as destination, then print

# current path[]

if u == d:

path2 = path.copy()

dist =0

for i in range(len(path2)-1) :

dist = dist + r[path2[i]][path2[i+1]]

#For all paths possible

if ev.route\_dist\_all[id][0] == 0:

ev.route\_dist\_all[id][0] = dist

ev.route\_all[id][0] = path2

else :

ev.route\_dist\_all[id].append(dist)

ev.route\_all[id].append(path2)

'''#Fortwo best paths of each car

if ev.route\_dist[id][0] == 0 and ev.route\_dist[id][1] == 0 :

ev.route\_dist[id][0] = dist

ev.route[id][0] = path2

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2

elif ev.route\_dist[id][0] > dist and ev.route\_dist[id][1] > dist:

ev.route\_dist[id][0] = dist

ev.route[id][0] = path2

elif ev.route[id][0]==ev.route[id][1]:

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2

elif ev.route\_dist[id][1] > dist :

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2'''

else:

# If current vertex is not destination

# Recur for all the vertices adjacent to this vertex

for i in self.graph[u]:

if visited[i]== False and r[u][i] < 100 \* ev.stats[id][4]:

self.printAllPathsUtil(i, d, visited, path, id)

# Remove current vertex from path[] and mark it as unvisited

path.pop()

visited[u]= False

# Prints all paths from 's' to 'd'

def printAllPaths(self, s, d, id):

# Mark all the vertices as not visited

visited =[False]\*(self.V)

# Create an array to store paths

path = []

# Call the recursive helper function to print all paths

self.printAllPathsUtil(s, d, visited, path, id)

t = Graph2(noCity)

for i in range(noCity):

for j in range(noCity) :

if r[i][j] < 10000 :

t.addEdge(i,j)

for id in range(noCar) :

t.printAllPaths(int(ev.stats[id][0]-1),int(ev.stats[id][1]-1),id)

#print(ev.route\_dist\_all)

## for natural form of path

'''for i in range(noCar) :

for j in range(2) :

ev.route[i][j] = [x+1 for x in ev.route[i][j]]'''

#print(ev.route\_all[0])

for id in range(noCar) :

sort\_arg = np.argsort(ev.route\_dist\_all[id])

for j in range(min(maxDepth,len(ev.route\_all[id]))) :

n = sort\_arg[j]

if ev.route\_dist[id][0] == 0:

ev.route\_dist[id][0] = ev.route\_dist\_all[id][n]

ev.route[id][0] = ev.route\_all[id][n]

else :

ev.route\_dist[id].append(ev.route\_dist\_all[id][n])

ev.route[id].append(ev.route\_all[id][n])

#print(ev.route\_dist)

## for natural form of path

'''for i in range(noCar) :

for j in range(2) :

ev.route[i][j] = [x+1 for x in ev.route[i][j]]'''

#print(ev.route)

batc = np.zeros((noCar,noCity), dtype=int)

battery\_needed = [0.]\*noCar

path = []

current\_battery = [0]\*noCar

visited = [[-1 for column in range(1)]

for row in range(noCity)]

#Assuming one path

for id in range(noCar):

path.append(ev.route[id][0])

#battery\_needed[id] = max(0,math.ceil(ev.route\_dist[id][0]/ev.stats[id][4] - ev.stats[id][2]))

current\_battery[id] = int(ev.stats[id][2])

path\_optimal = []

batc\_optimal = np.zeros((noCar,noCity), dtype=int)

time\_optimal = [[0 for column in range(1)]

for row in range(noCar)]

minimum\_time\_achieved = sys.maxsize

charge\_start\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

charge\_end\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

def final\_time (bc,path) :

time = [[0 for column in range(1)]

for row in range(noCar)]

charging\_start\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

charging\_end\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

for id in range(noCar) :

for j in range(len(path[id])) :

if j==0:

time[id][0] = 0

else :

time[id].append(time[id][j-1]+r[path[id][j-1]][path[id][j]]/ev.stats[id][6])

instances = [0]\*noCity

car\_at\_instance = [[-1 for column in range(1)]

for row in range(noCity)]

for i in range(noCity) :

for j in range(noCar) :

if bc[j][i] > 0 :

instances[i] += 1

if car\_at\_instance[i][0] == -1:

car\_at\_instance[i][0] = j

else:

car\_at\_instance[i].append(j)

#print("instances",instances)

#print("car\_instances",car\_at\_instance)

#Adding the charging time to all

for i in range(noCity) :

if instances[i] >= 1 :

for j in range(len(car\_at\_instance[i])) :

id = car\_at\_instance[i][j]

if id > -1 :

count = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

charging\_start\_time[id][i] = time[id][k]

charging\_end\_time[id][i] = time[id][k] + bc[id][i]\*ev.stats[id][3]

if count>0 :

time[id][k] = time[id][k] + bc[id][i]\*ev.stats[id][3]

visited = car\_at\_instance.copy()

#Adding the delay time if any:

def delay() :

counter = 0

for i in range(noCity) :

if instances[i] > 1 :

max\_time = 0

max\_time\_car = -1

id = -1

#time\_stamp = [0]\*len(visited[i])

j\_val = -1

for j in range(len(visited[i])) :

id = visited[i][j]

if visited[i][-2] == -1 :

max\_time = time[id][-1]

max\_time\_car = id

j\_val = j

else :

if time[id][-1] > max\_time and id > -1:

max\_time = time[id][-1]

max\_time\_car = id

j\_val = j

counter = 1

if id==(noCity-1) and counter == 0 :

return 0

for j in range(len(car\_at\_instance[i])) :

id = visited[i][j]

if id > -1 and id != max\_time\_car:

if charging\_end\_time[id][i] < charging\_start\_time[max\_time\_car][i] :

visited[i][j\_val] = -1

return 1

elif charging\_end\_time[max\_time\_car][i] < charging\_start\_time[id][i] :

visited[i][j\_val] = -1

return 1

else :

if charging\_start\_time[id][i] < charging\_start\_time[max\_time\_car][i]:

count = 0

diff = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

diff = min(charging\_end\_time[max\_time\_car][i]-charging\_start\_time[max\_time\_car][i], abs(charging\_end\_time[id][i]-charging\_start\_time[max\_time\_car][i]))

charging\_end\_time[id][i] = charging\_end\_time[id][i] + diff

if count>0 :

time[id][k] = time[id][k] + diff

visited[i][j\_val] = -1

return 1

elif charging\_start\_time[id][i] < charging\_end\_time[max\_time\_car][i]:

count = 0

diff = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

diff = charging\_end\_time[max\_time\_car][i]-charging\_start\_time[id][i]

charging\_end\_time[id][i] = charging\_end\_time[id][i] + diff

charging\_start\_time[id][i] = charging\_end\_time[max\_time\_car][i]

if count>0 :

time[id][k] = time[id][k] + diff

visited[i][j\_val] = -1

return 1

elif i == noCar-1 and counter == 0:

return 0

while True :

i = delay()

if i == 0 :

break

max\_Total\_Time = 0

for id in range(noCar) :

if time[id][-1] > max\_Total\_Time :

max\_Total\_Time = time[id][-1]

global path\_optimal

global batc\_optimal

global time\_optimal

global charge\_end\_time

global charge\_start\_time

global minimum\_time\_achieved

if max\_Total\_Time < minimum\_time\_achieved :

#print("Entered")

minimum\_time\_achieved = max\_Total\_Time

path\_optimal = path.copy()

batc\_optimal = batc.copy()

time\_optimal = time.copy()

charge\_start\_time = charging\_start\_time.copy()

charge\_end\_time = charging\_end\_time.copy()

def time\_calculator(id, src, batr,current,path) :

## here reduced src form

##batr = battery remaining

## if src in path is 2 read 1

if id==(noCar-1) and src==path[id][-2] :

#print(batc)

final\_time(batc,path)

else :

if src==path[id][-2] :

time\_calculator(id+1,int(ev.stats[id+1][0]-1), battery\_needed[id+1],current\_battery[id+1],path)

else :

for i in range(len(path[id])-1):

if path[id][i] == src :

next = path[id][i+1]

batRangeMax = int(min(batr,100-current))

batRangeMin = int(max(0, math.ceil(r[src][next]/ev.stats[id][4] - current)))

for j in range(batRangeMin,batRangeMax+1) :

if next == path[id][-2]:

if ( batr + current - r[src][next]/ev.stats[id][4] ) > 100 :

return

else :

batc[id][src] = j

batc[id][next] = batr - j

time\_calculator(id, next, batr - j,current + j - r[src][next]/ev.stats[id][4],path)

else:

batc[id][src] = j

time\_calculator(id, next, batr-j,current + j - r[src][next]/ev.stats[id][4],path)

counter\_for\_initialisation = 0

def path\_builder(id) :

global battery\_needed

if id == (noCar - 1) :

time\_calculator(0,int(ev.stats[0][0])-1,battery\_needed[0],current\_battery[0],path)

#print(path)

else:

for j in range(len(ev.route[id])) :

path[id] = ev.route[id][j]

battery\_needed[id] = max(0,math.ceil(ev.route\_dist[id][j]/ev.stats[id][4] - ev.stats[id][2]))

#print(ev.route[id][j])

path\_builder(id + 1)

path\_builder(0)

#time\_calculator(0,int(ev.stats[0][0])-1,battery\_needed[0],current\_battery[0])

print("Path Taken",path\_optimal)

print("Battery charge Matrix",batc\_optimal)

print("Time charging starts of car i at node j",charge\_start\_time)

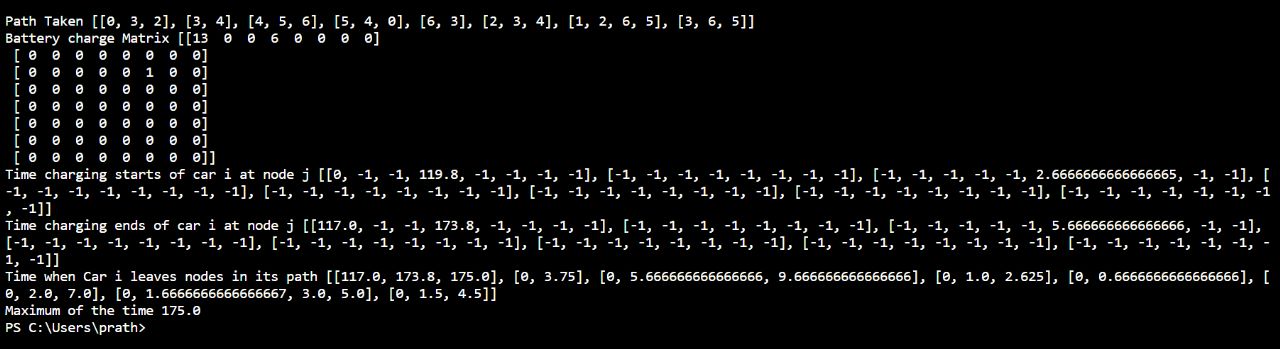
print("Time charging ends of car i at node j",charge\_end\_time)

print("Time when Car i leaves nodes in its path",time\_optimal)

print("Maximum of the time",minimum\_time\_achieved)

**RESULT**

**Result for the Source Code of Existing System:**



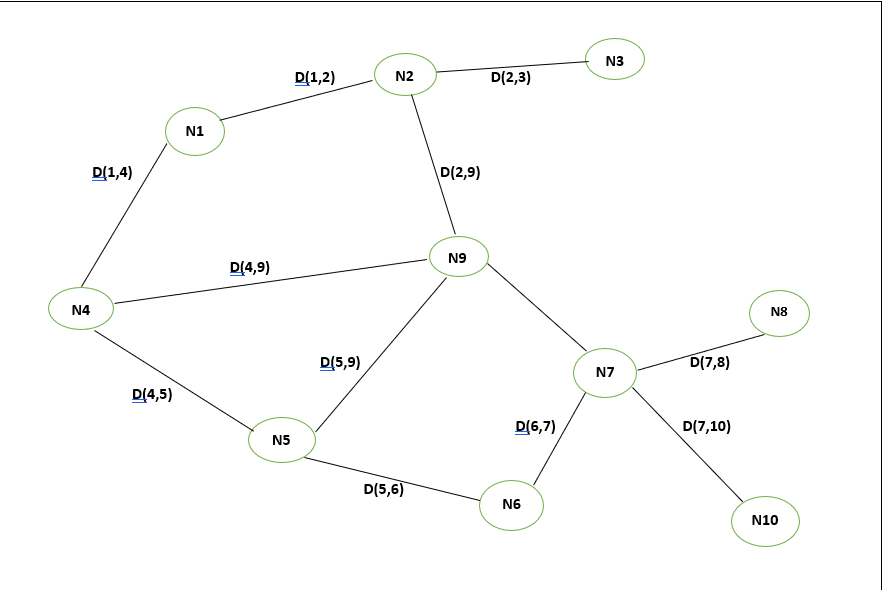
**CHAPTER 4**

**PROPOSING A NEW SYSTEM**

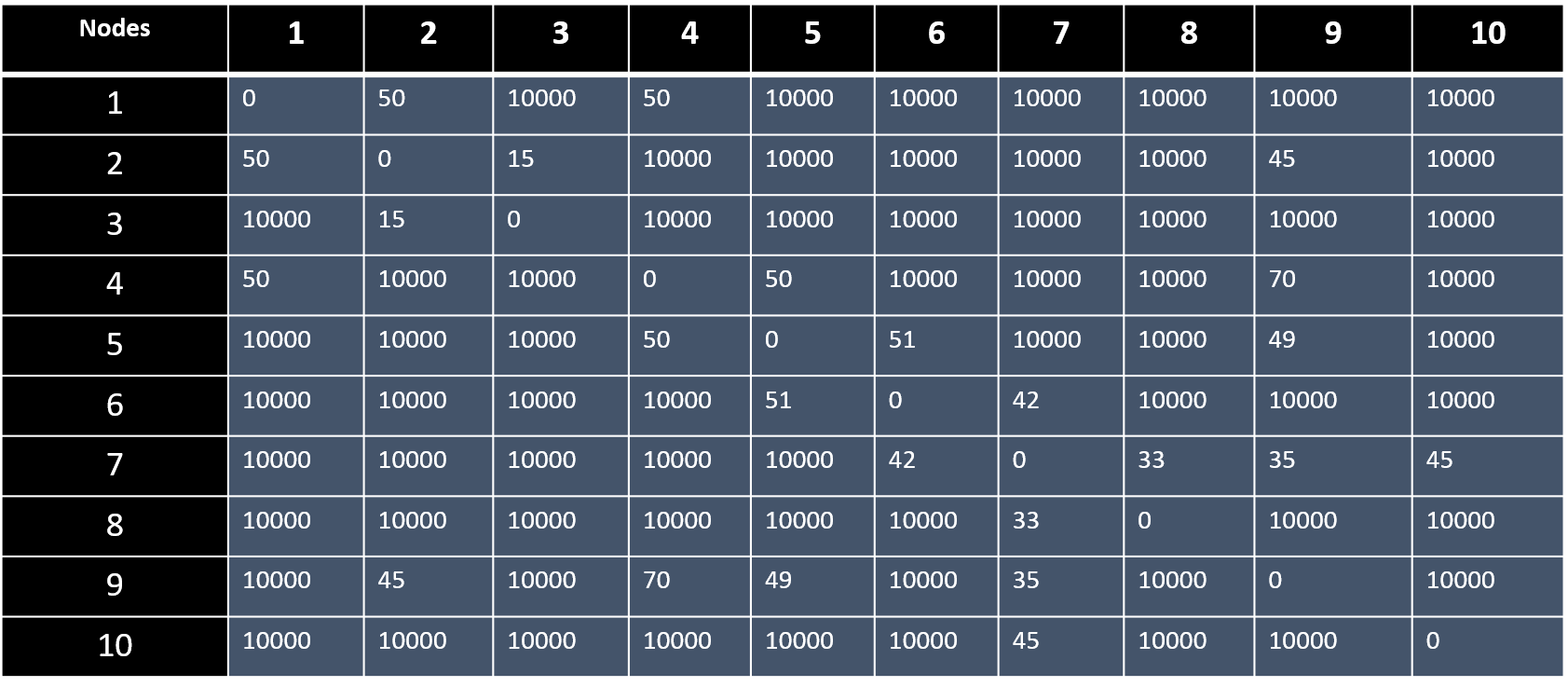
Consider an area network of the adopted suburban area in Hyderabad city where we need to route a set of electric vehicles which may require to be charged during its journey from some source to some destination. Assume that each city has a single charging station which can charge one EV at a time. Consider a set of 10 EVs namely EV1, EV2, . . . ,EV10. For each EV the following information is provided.

1. source node
2. destination node
3. battery charge status initially
4. charging rate for battery at a charging station (percent charged per unit time)
5. discharging rate of battery while traveling (distance travel per unit charge)
6. maximum battery capacity
7. average traveling speed (distance per unit time).

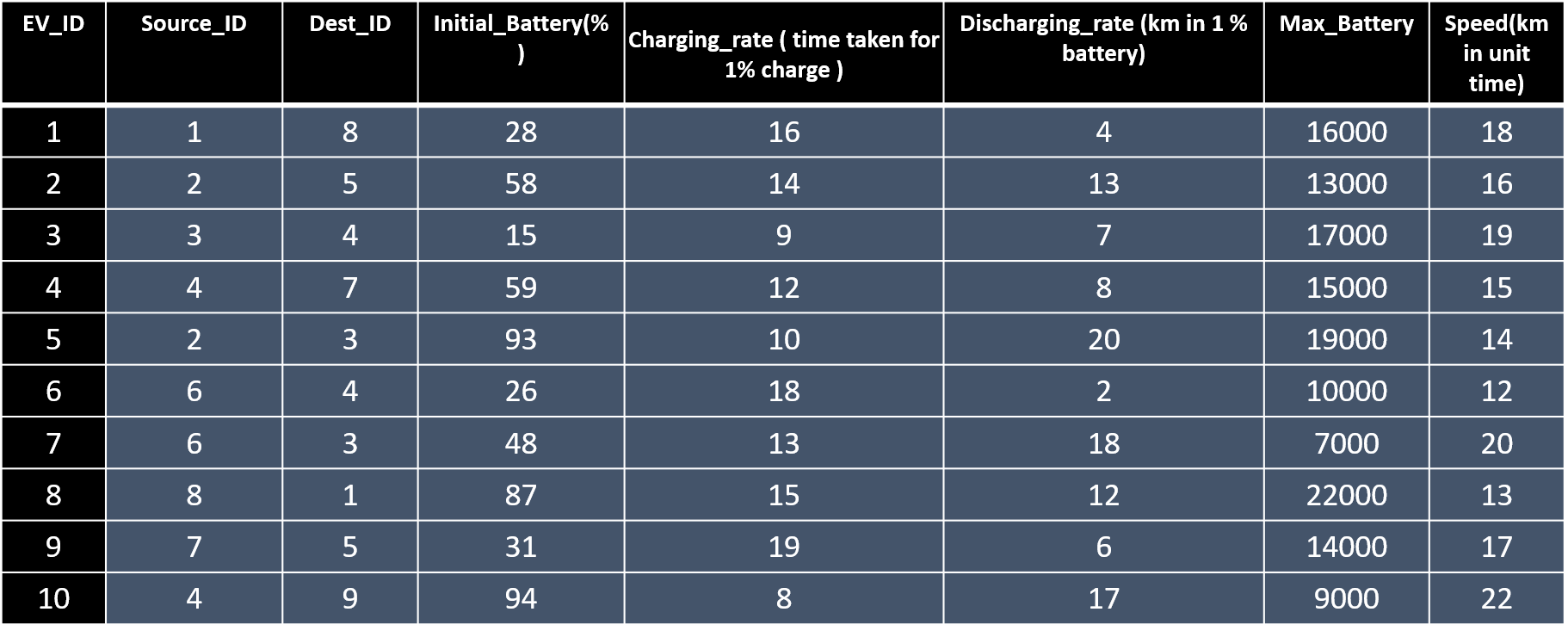
Assume that all vehicles start their journey at t = 0 and EV r reaches it’s destination at t = Tr. We need to route all the vehicles from their respective sources to destinations such that max{Tr} is minimized.

**AREA NETWORK:**

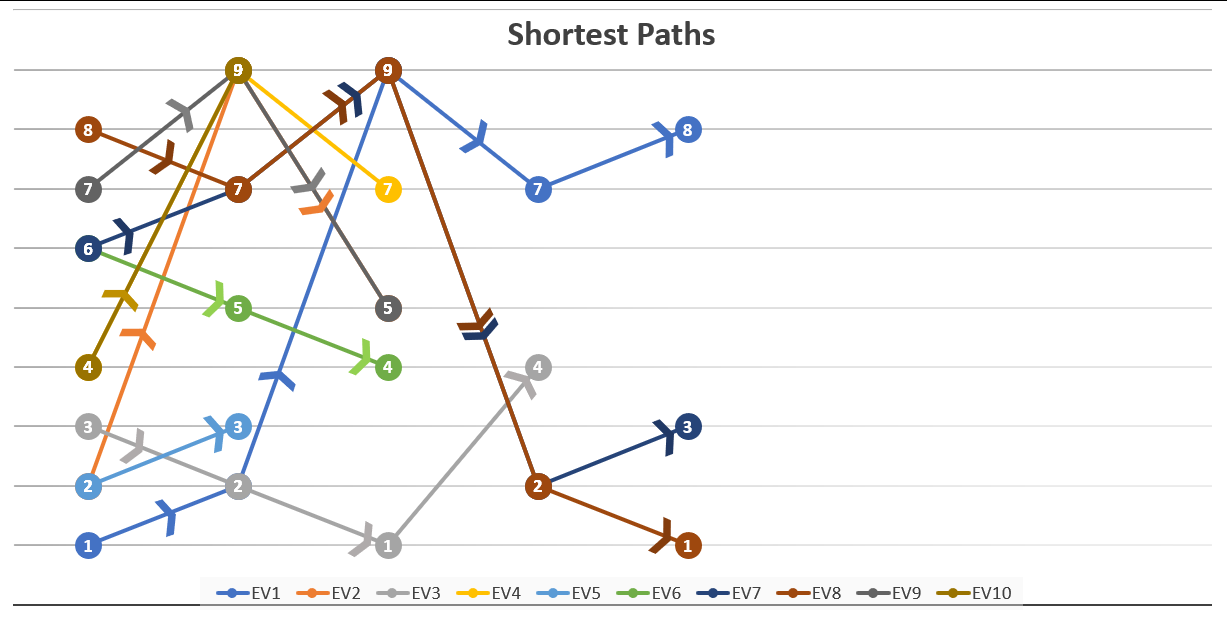
**NODE TO NODE DISTANCES:**



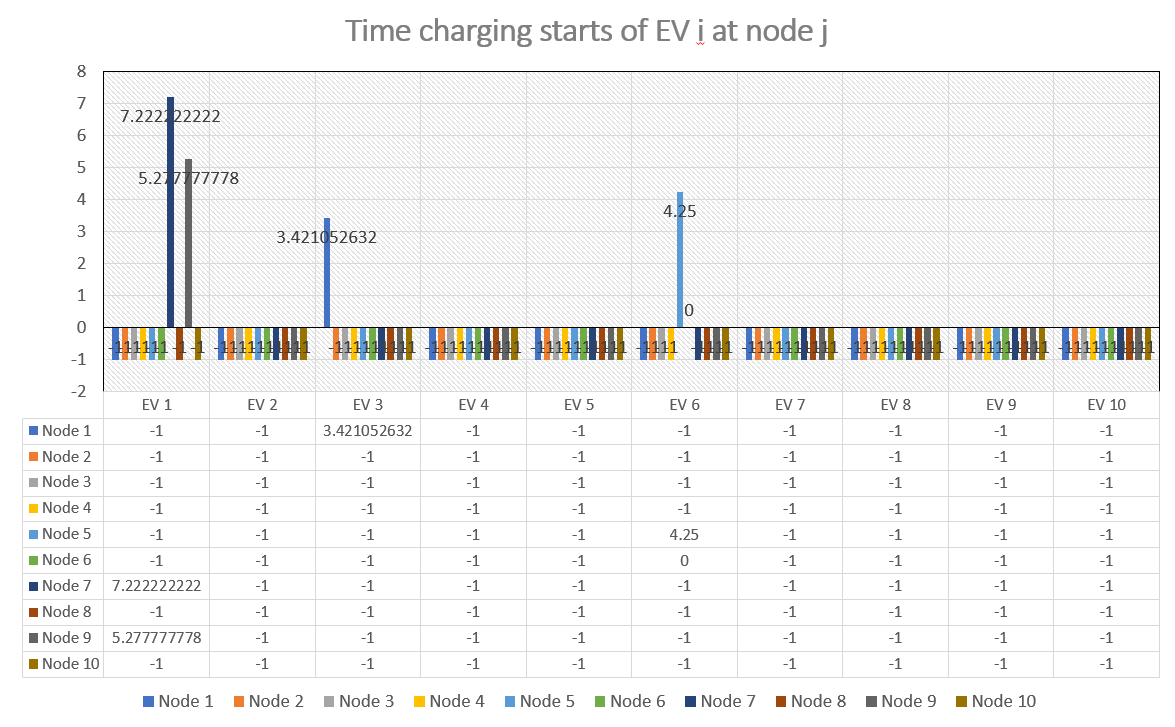
**EV DETAILS:**



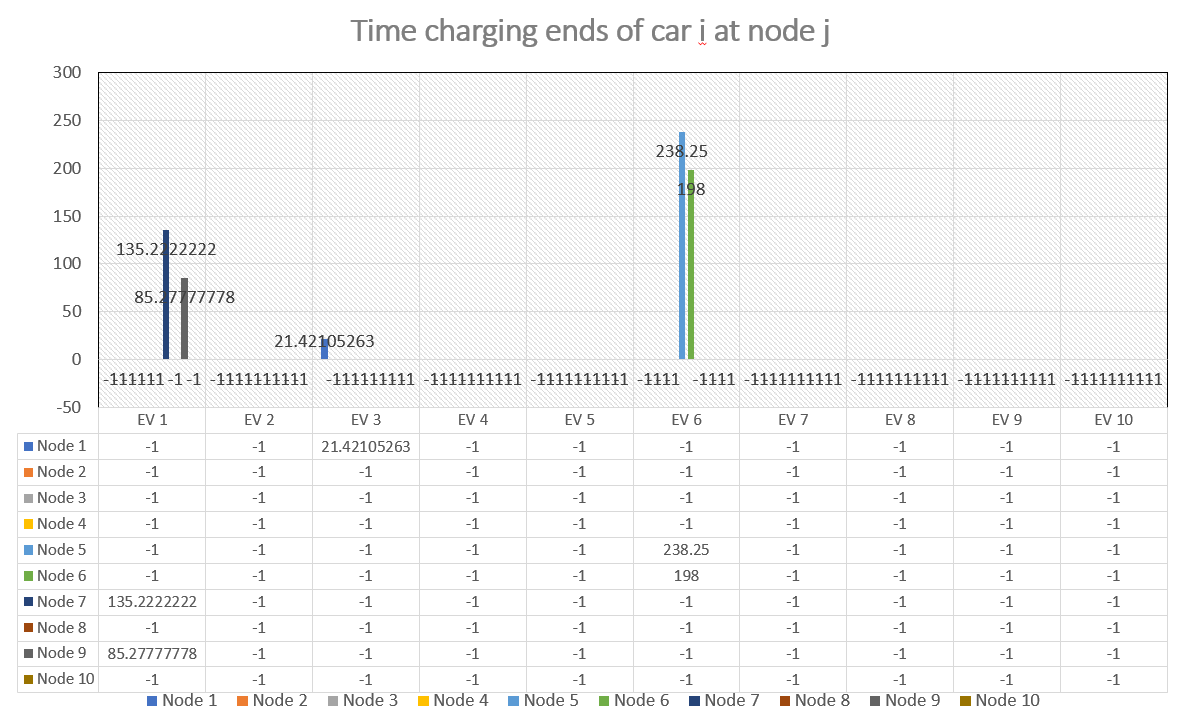
**The below graph depicts the paths that are to be taken by the EVs:**



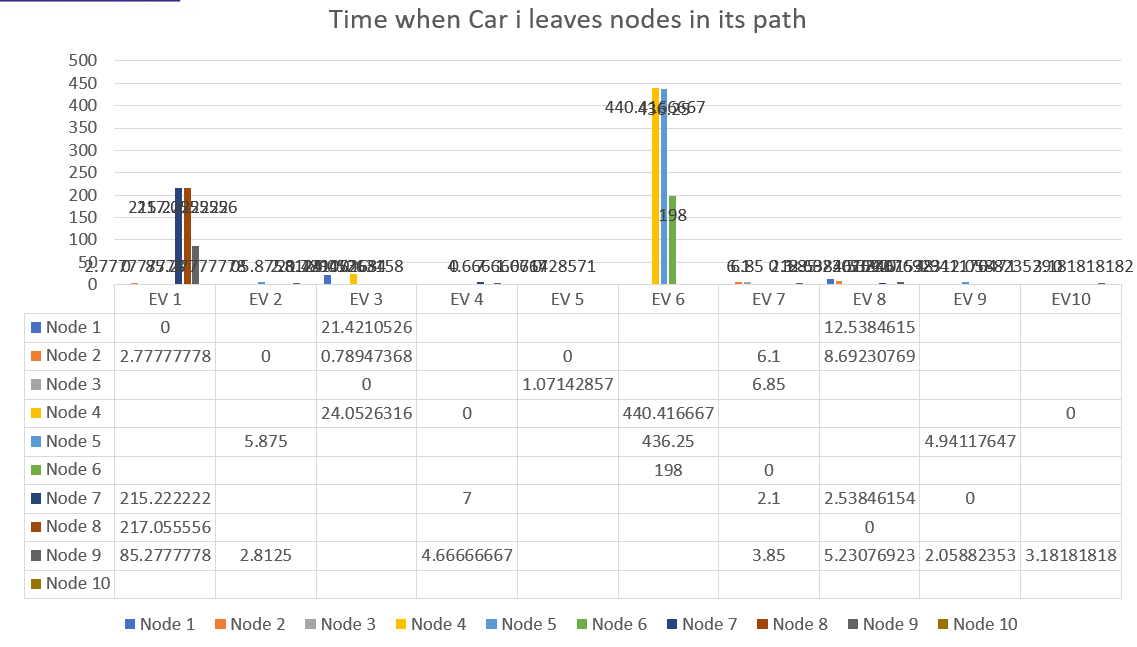
**The below graph depicts at what time the battery starts charging:**

****

**The below graph depicts at what time the battery ends charging:**

****

**The below graph depicts at what time EV leaves a particular node:**

****

This the proposed system with reference to the above already existing system.

**SOURCE CODE OF PROPOSED SYSTEM**

import sys

import numpy as np

import math

noCity = 10

noCar = 10

maxDepth = 1

class Cars():

def \_init\_(self, quantity):

self.V = quantity

self.stats = [[0 for column in range(15)]

for row in range(quantity)]

self.route\_all = [[0 for column in range(1)]

for row in range(quantity)]

self.route\_dist\_all = [[0 for column in range(1)]

for row in range(quantity)]

self.route = [[0 for column in range(1)]

for row in range(quantity)]

self.route\_dist = [[0 for column in range(1)]

for row in range(quantity)]

ev = Cars(noCar)

q = np.genfromtxt(r"C:\Users\prath\OneDrive\Desktop\EV PROJECT 1\EV.csv", delimiter=',')

q = q.tolist()

ev.stats = q

r = np.genfromtxt(r"C:\Users\prath\OneDrive\Desktop\EV PROJECT 1\Nodes.csv", delimiter=',')

r = r.tolist()

r[0][0] = 0

# Python program to print all paths from a source to destination.

from collections import defaultdict

# This class represents a directed graph

# using adjacency list representation

class Graph2:

def \_init\_(self, vertices):

# No. of vertices

self.V = vertices

# default dictionary to store graph

self.graph = defaultdict(list)

# function to add an edge to graph

def addEdge(self, u, v):

self.graph[u].append(v)

'''A recursive function to print all paths from 'u' to 'd'.

visited[] keeps track of vertices in current path.

path[] stores actual vertices and path\_index is current

index in path[]'''

def printAllPathsUtil(self, u, d, visited, path, id):

# Mark the current node as visited and store in path

visited[u]= True

path.append(u)

# If current vertex is same as destination, then print

# current path[]

if u == d:

path2 = path.copy()

dist =0

for i in range(len(path2)-1) :

dist = dist + r[path2[i]][path2[i+1]]

#For all paths possible

if ev.route\_dist\_all[id][0] == 0:

ev.route\_dist\_all[id][0] = dist

ev.route\_all[id][0] = path2

else :

ev.route\_dist\_all[id].append(dist)

ev.route\_all[id].append(path2)

'''#Fortwo best paths of each car

if ev.route\_dist[id][0] == 0 and ev.route\_dist[id][1] == 0 :

ev.route\_dist[id][0] = dist

ev.route[id][0] = path2

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2

elif ev.route\_dist[id][0] > dist and ev.route\_dist[id][1] > dist:

ev.route\_dist[id][0] = dist

ev.route[id][0] = path2

elif ev.route[id][0]==ev.route[id][1]:

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2

elif ev.route\_dist[id][1] > dist :

ev.route\_dist[id][1] = dist

ev.route[id][1] = path2'''

else:

# If current vertex is not destination

# Recur for all the vertices adjacent to this vertex

for i in self.graph[u]:

if visited[i]== False and r[u][i] < 100 \* ev.stats[id][4]:

self.printAllPathsUtil(i, d, visited, path, id)

# Remove current vertex from path[] and mark it as unvisited

path.pop()

visited[u]= False

# Prints all paths from 's' to 'd'

def printAllPaths(self, s, d, id):

# Mark all the vertices as not visited

visited =[False]\*(self.V)

# Create an array to store paths

path = []

# Call the recursive helper function to print all paths

self.printAllPathsUtil(s, d, visited, path, id)

t = Graph2(noCity)

for i in range(noCity):

for j in range(noCity) :

if r[i][j] < 10000 :

t.addEdge(i,j)

for id in range(noCar) :

t.printAllPaths(int(ev.stats[id][0]-1),int(ev.stats[id][1]-1),id)

#print(ev.route\_dist\_all)

## for natural form of path

'''for i in range(noCar) :

for j in range(2) :

ev.route[i][j] = [x+1 for x in ev.route[i][j]]'''

#print(ev.route\_all[0])

for id in range(noCar) :

sort\_arg = np.argsort(ev.route\_dist\_all[id])

for j in range(min(maxDepth,len(ev.route\_all[id]))) :

n = sort\_arg[j]

if ev.route\_dist[id][0] == 0:

ev.route\_dist[id][0] = ev.route\_dist\_all[id][n]

ev.route[id][0] = ev.route\_all[id][n]

else :

ev.route\_dist[id].append(ev.route\_dist\_all[id][n])

ev.route[id].append(ev.route\_all[id][n])

#print(ev.route\_dist)

## for natural form of path

'''for i in range(noCar) :

for j in range(2) :

ev.route[i][j] = [x+1 for x in ev.route[i][j]]'''

#print(ev.route)

batc = np.zeros((noCar,noCity), dtype=int)

battery\_needed = [0.]\*noCar

path = []

current\_battery = [0]\*noCar

visited = [[-1 for column in range(1)]

for row in range(noCity)]

#Assuming one path

for id in range(noCar):

path.append(ev.route[id][0])

#battery\_needed[id] = max(0,math.ceil(ev.route\_dist[id][0]/ev.stats[id][4] - ev.stats[id][2]))

current\_battery[id] = int(ev.stats[id][2])

path\_optimal = []

batc\_optimal = np.zeros((noCar,noCity), dtype=int)

time\_optimal = [[0 for column in range(1)]

for row in range(noCar)]

minimum\_time\_achieved = sys.maxsize

charge\_start\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

charge\_end\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

def final\_time (bc,path) :

time = [[0 for column in range(1)]

for row in range(noCar)]

charging\_start\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

charging\_end\_time = [[-1 for column in range(noCity)]

for row in range(noCar)]

for id in range(noCar) :

for j in range(len(path[id])) :

if j==0:

time[id][0] = 0

else :

time[id].append(time[id][j-1]+r[path[id][j-1]][path[id][j]]/ev.stats[id][6])

instances = [0]\*noCity

car\_at\_instance = [[-1 for column in range(1)]

for row in range(noCity)]

for i in range(noCity) :

for j in range(noCar) :

if bc[j][i] > 0 :

instances[i] += 1

if car\_at\_instance[i][0] == -1:

car\_at\_instance[i][0] = j

else:

car\_at\_instance[i].append(j)

#print("instances",instances)

#print("car\_instances",car\_at\_instance)

#Adding the charging time to all

for i in range(noCity) :

if instances[i] >= 1 :

for j in range(len(car\_at\_instance[i])) :

id = car\_at\_instance[i][j]

if id > -1 :

count = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

charging\_start\_time[id][i] = time[id][k]

charging\_end\_time[id][i] = time[id][k] + bc[id][i]\*ev.stats[id][3]

if count>0 :

time[id][k] = time[id][k] + bc[id][i]\*ev.stats[id][3]

visited = car\_at\_instance.copy()

#Adding the delay time if any:

def delay() :

counter = 0

for i in range(noCity) :

if instances[i] > 1 :

max\_time = 0

max\_time\_car = -1

id = -1

#time\_stamp = [0]\*len(visited[i])

j\_val = -1

for j in range(len(visited[i])) :

id = visited[i][j]

if visited[i][-2] == -1 :

max\_time = time[id][-1]

max\_time\_car = id

j\_val = j

else :

if time[id][-1] > max\_time and id > -1:

max\_time = time[id][-1]

max\_time\_car = id

j\_val = j

counter = 1

if id==(noCity-1) and counter == 0 :

return 0

for j in range(len(car\_at\_instance[i])) :

id = visited[i][j]

if id > -1 and id != max\_time\_car:

if charging\_end\_time[id][i] < charging\_start\_time[max\_time\_car][i] :

visited[i][j\_val] = -1

return 1

elif charging\_end\_time[max\_time\_car][i] < charging\_start\_time[id][i] :

visited[i][j\_val] = -1

return 1

else :

if charging\_start\_time[id][i] < charging\_start\_time[max\_time\_car][i]:

count = 0

diff = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

diff = min(charging\_end\_time[max\_time\_car][i]-charging\_start\_time[max\_time\_car][i], abs(charging\_end\_time[id][i]-charging\_start\_time[max\_time\_car][i]))

charging\_end\_time[id][i] = charging\_end\_time[id][i] + diff

if count>0 :

time[id][k] = time[id][k] + diff

visited[i][j\_val] = -1

return 1

elif charging\_start\_time[id][i] < charging\_end\_time[max\_time\_car][i]:

count = 0

diff = 0

for k in range(len(path[id])) :

if path[id][k] == i :

count = 1

diff = charging\_end\_time[max\_time\_car][i]-charging\_start\_time[id][i]

charging\_end\_time[id][i] = charging\_end\_time[id][i] + diff

charging\_start\_time[id][i] = charging\_end\_time[max\_time\_car][i]

if count>0 :

time[id][k] = time[id][k] + diff

visited[i][j\_val] = -1

return 1

elif i == noCar-1 and counter == 0:

return 0

while True :

i = delay()

if i == 0 :

break

max\_Total\_Time = 0

for id in range(noCar) :

if time[id][-1] > max\_Total\_Time :

max\_Total\_Time = time[id][-1]

global path\_optimal

global batc\_optimal

global time\_optimal

global charge\_end\_time

global charge\_start\_time

global minimum\_time\_achieved

if max\_Total\_Time < minimum\_time\_achieved :

#print("Entered")

minimum\_time\_achieved = max\_Total\_Time

path\_optimal = path.copy()

batc\_optimal = batc.copy()

time\_optimal = time.copy()

charge\_start\_time = charging\_start\_time.copy()

charge\_end\_time = charging\_end\_time.copy()

def time\_calculator(id, src, batr,current,path) :

## here reduced src form

##batr = battery remaining

## if src in path is 2 read 1

if id==(noCar-1) and src==path[id][-2] :

#print(batc)

final\_time(batc,path)

else :

if src==path[id][-2] :

time\_calculator(id+1,int(ev.stats[id+1][0]-1), battery\_needed[id+1],current\_battery[id+1],path)

else :

for i in range(len(path[id])-1):

if path[id][i] == src :

next = path[id][i+1]

batRangeMax = int(min(batr,100-current))

batRangeMin = int(max(0, math.ceil(r[src][next]/ev.stats[id][4] - current)))

for j in range(batRangeMin,batRangeMax+1) :

if next == path[id][-2]:

if ( batr + current - r[src][next]/ev.stats[id][4] ) > 100 :

return

else :

batc[id][src] = j

batc[id][next] = batr - j

time\_calculator(id, next, batr - j,current + j - r[src][next]/ev.stats[id][4],path)

else:

batc[id][src] = j

time\_calculator(id, next, batr-j,current + j - r[src][next]/ev.stats[id][4],path)

counter\_for\_initialisation = 0

def path\_builder(id) :

global battery\_needed

if id == (noCar - 1) :

time\_calculator(0,int(ev.stats[0][0])-1,battery\_needed[0],current\_battery[0],path)

#print(path)

else:

for j in range(len(ev.route[id])) :

path[id] = ev.route[id][j]

battery\_needed[id] = max(0,math.ceil(ev.route\_dist[id][j]/ev.stats[id][4] - ev.stats[id][2]))

#print(ev.route[id][j])

path\_builder(id + 1)

path\_builder(0)

#time\_calculator(0,int(ev.stats[0][0])-1,battery\_needed[0],current\_battery[0])

print("Path Taken",path\_optimal)

print("Battery charge Matrix",batc\_optimal)

print("Time charging starts of car i at node j",charge\_start\_time)

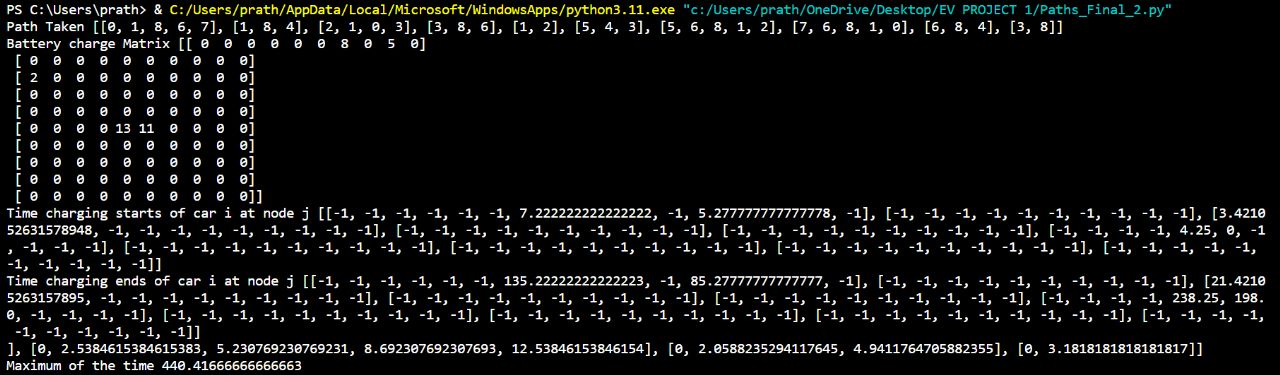
print("Time charging ends of car i at node j",charge\_end\_time)

print("Time when Car i leaves nodes in its path",time\_optimal)

print("Maximum of the time",minimum\_time\_achieved)

**RESULT**

**Result for the Source Code of Proposed System:**



This is the result obtained by simulating the proposed system. Here in proposed system we considered an area network of Hyderabad locality with 10 nodes and 10 electric vehicles. On a total of 3 vehicles gets charged out of 10 vehicles taken. Also in the result obtained we can find the time taken by the vehicle to charge it’s battery at a particular charging station and the time at which the electric vehicle starts charging and ending.

**CHAPTER 7**

**CONCLUSION**

This project demonstrates that planning a path to electric vehicles has more advantages than conventional routing methods, that merely consider time or distance to the destination into account. Three factors make up the multi objective optimization: battery lifetime, travel time, and energy consumption. With different solutions are produced by applying different weights to the optimization variables. Testing is done for current street systems. No wastage of time for charging the battery with more than the required amount of charge to reach the destination will be observed here. So that during any emergency a very less delay will be made here in this scenario.

**REFRENCES**

* Li, H., Alsolami, M., Yang, S., Alsmadi, Y.M., Wang, J. (2017). Lifetime test design for second-use electric vehicle batteries in residential applications. IEEE Transactions on Sustainable Energy, 8(4), 1736–1746. https://doi.org/10.1109/TSTE.2017.2707565.
* Yi, Z., & Bauer, P.H. (2017). Adaptive multiresolution energy consumption prediction for electric vehicles. IEEE Trans- actions on Vehicular Technology, 66(11), 10515–10525. <https://doi.org/10.1109/TVT.2017.2720587>.
* Nunzio, G.D., & Thibault, L. (2017). Energy-optimal driving range prediction for electric vehicles. In 2017 IEEE Intelligent Vehicles Symposium (IV) (pp. 1608–1613). https://doi. org/10.1109/IVS.2017.7995939.
* Gota, S., Huizenga, C., Peet, K., Medimorec, N., Bakker, S. (2019). Decarbonising transport to achieve paris agreement targets. Energy Efficiency, 12(2), 363–386. <https://doi.org/10.1007/s12053-018-9671-3>.
* Song, Z., Duan, H., Zhou, S., Qiu, X. (2015). Urban route planning considering traffic flows. In: 2015 Chinese Automation Congress (CAC), pp. 1940–1944. https://doi.org/10.11 09/CAC.2015.7382822.