1.Importing necessary libraries

```
[ ]: import numpy as np
from collections import defaultdict
import pandas as pd
import itertools
import argparse
```

2.Taking User Input from terminal

i. Number of cars
ii.Details of car: source of car , destination of car , initial battery of car , Maximum battery capacity of car , charging rate of car , discharging rate of car , Average speed of car

ii. Number of cities

iii. Pair of cities (i.e the two cities which are linked) and distance between them.

```
[ ]: parser = argparse.ArgumentParser() #Arg parser to get input
parser.add_argument('--n', dest='nodes',nargs=3,action='append',help="details␣
↪of graph node-1 node-2 distance between them")
parser.add_argument('--num_cars',type=int,required=True,help='No of Cars')
parser.add_argument('--c', dest='cars',nargs=7,action='append',help="details of␣
↪cars in following order source of car , destination of car , init_battery of␣
↪car , Max_battery of car , charging_rate of car , discharging_rate of car ,␣
↪Average speed of car ")
args = parser.parse_args()
```

3.Using user input, a map of cities is made with cities as vertices and distance between them as edges

```
[ ]: Graphs = {}
for node_1, node_2, dis in args.nodes:
  if node_1 in Graphs.keys():
    values,distances = (Graphs[node_1])[0],(Graphs[node_1])[1]
    values.append(int(node_2))
    distances.append(float(dis))
    Graphs[node_1] = [values,distances]
  else:
    Graphs[node_1] = [[int(node_2)],[float(dis)]]
```

```
  if node_2 in Graphs.keys():
    values,distances = (Graphs[node_2])[0],(Graphs[node_2])[1]
    values.append(int(node_1))
    distances.append(float(dis))
    Graphs[node_2] = [values,distances]
  else:
    Graphs[node_2] = [[int(node_1)],[float(dis)]]
```

4.Car class with different functions:

i. **init** :initialises the variables of a car like source ,destination, initial battery status, Maximum battery capacity ,charging rate,discharging rate, Average speed

ii. update: updates the variables associated with car like current location, travelling time, battery staus and the boolean variable charging which is True is the car requires charging at that city otherwise false

iii. charging: accounts for time required for self charge and also updates boolean variable charging which may be used by other cars.

iv. enough_battery:checks whether car has sufficient battery to move to next city or not

v. wait: accounts for time required in waiting while the other cars at that city get charged, since only one car recharge its battery in a city at a time

[ ]:
```
class Car(): #Car Class to store all variables related to car like␣
↪source,destination,tr,current_location,path
  def␣
↪__init__(self,source,destination,init_battery,Max_battery,charging_rate,discharging_rate,sp
↪
    self.source = source
    self.destination = destination
    self.current_location = source
    self.battery_status = init_battery
    self.max_battery = Max_battery
    self.charging_rate = charging_rate
    self.discharging_rate = discharging_rate
    self.speed = speed
    self.tr = 0
    self.path = []
    self.path.append(self.source)
    self.charging_ = False

  def update(self,new_location,distance):
    self.current_location = str(new_location)
    self.tr = self.tr + distance/self.speed
    self.battery_status = self.battery_status - (distance/self.speed)*self.
↪discharging_rate
    self.path.append(new_location)
```

```
    self.charging_ = False
  def charging(self):
    self.tr =  self.tr + (self.max_battery- self.battery_status)/(self.
 ↪charging_rate)
    self.battery_status = self.max_battery
    self.charging_ = True

  def enough_battery(self,distance):
    if self.battery_status >= (distance/self.speed)*self.discharging_rate:
      return True
    else:
      return False
  def wait(self,all_cars,current_car):
    all_cars.remove(current_car)
    for car in all_cars:
      if car.current_location ==  current_car.current_location and car.charging:
        self.tr = self.tr + (car.max_battery- car.battery_status)/(car.
 ↪charging_rate)
        break
```

5.Other functions-

i. check_car: checks whether and which other cars are there in the current city that require charging

ii. remove_repeated_ones: makes sure that a car doesn't visit a city more than once

iii.get_cars :Function to get initial all cars to user defined conditions

[ ]:
```
def check_car(all_cars,current_car):
  all_cars.remove(current_car)
  for car in all_cars:
    if car.current_location ==  current_car.current_location and car.charging:
      return True
  return False

def remove_repeated_ones(childs,childs_dis,path):
  for child in childs:
    if child in path:
      index = childs.index(child)
      childs.remove(child)
      childs_dis.remove(childs_dis[index])

  return childs,childs_dis

def get_cars(args): #Function to get initial all cars to user defined␣
 ↪conditions
        cars_all = []
        for i in range(args.num_cars):
```

```
        source              =  args.cars[i][0]
        destination         = args.cars[i][1]
        init_battery        = float(args.cars[i][2])
        Max_battery         = float(args.cars[i][3])
        charging_rate       = float(args.cars[i][4])
        discharging_rate    = float(args.cars[i][5])
        speed               = float(args.cars[i][6])
        car_ = Car(source,destination,init_battery,Max_battery,charging_rate,discharging_rate,speed)
        cars_all.append(car_)
    return cars_all
```

6.main function
This is the implementation of depth first search with nearest neighbour search heuristic

i. The objects cars are stored in a list

ii.Using a loop, in each iteration a car is taken from the list and one move is made using the following steps:

(a)if current node is same as destination node, the node is added to path and the car is remove

(b)The node which is at minimum distance to the current node is chosen

(c)using enough_battery function , it is determined whether it is possible for the car to go to

(d)If it is not possible, then using check_cars function, it is determined whether there are ot there or not

(e)If there are , using wait function the waiting time is added to total time of car

(f)using charging function, the charging time is added to total time of car

iii. When the list becomes empty, the program terminates

[ ]:
```
cars = get_cars(args)
not_destination = True
reached_destination_all_cars = [0]*len(cars)
best_path = []
best_path_time = []
while cars!=[]:
  for car in cars:
    current_node  = car.current_location
    childs,childs_dis = Graphs[current_node]     #Childs of current node
    childs,childs_dis = remove_repeated_ones(childs,childs_dis,car.path)
    choosen_child_index = childs_dis.index(min(childs_dis))
    choosen_child,choosen_child_dis = childs[choosen_child_index],childs_dis[choosen_child_index]
    if car.enough_battery(choosen_child_dis):
```

```
      car.update(choosen_child,choosen_child_dis)
    else:
      if not check_car(cars,car):
        car.charging()
      else:
        car.wait(cars)

    for car in cars:
      if car.current_location==car.destination:
        best_path.append(car.path)
        best_path_time.append(car.tr)
        cars.remove(car)

print('By Heuristic Method:')
print('Best path: ', best_path)
print('Best path Time: ', best_path_time)
print('Max Tr: ', max(best_path_time))
```