Optimal_Algorithm

March 28, 2021

0.1 Importing the required libraries

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

This is the method used to obtain information from user like no of cars, all the details about car i.e. 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, no of cities, the pair of cities (i.e the two cities which are linked) and distance between them.

Note: Each pair of cities is unique i.e if tow cities A,B are connected to each other then the user has to only provide one pair either A,B or B,A and not both

[ ]:
```
parser = argparse.ArgumentParser() #Arg parser to get input
parser.add_argument('--n',␣
 ↪dest='nodes',nargs=3,required=True,action='append',help="details of graph␣
 ↪node-1 node-2 distance between them")
parser.add_argument('--Pruning',default=1,type=int,help='Run Algo With pruning␣
 ↪or without pruning')
parser.add_argument('--num_cars',type=int,required=True,help='No of Cars')
parser.add_argument('--c',␣
 ↪dest='cars',nargs=7,required=True,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()
```

Graph Class is used to form the map between all the cities and will be used in algorithm to get all possible paths from source to destination for a particular car

[ ]:
```
class Graph: #Graph to get all possible paths from source to destination for a␣
 ↪particular car
  def __init__(self, vertices):
    # No. of vertices
    self.V = vertices

        # default dictionary to store graph
```

```python
    self.graph = defaultdict(list)
    self.all_paths = []

    # function to add an edge to graph
  def addEdge(self, u, v):
    self.graph[u].append(v)

  '''A recursive function to get all paths from source ('u') to␣
↪Destination('d'). visited[] keeps track of vertices in current path. path[]␣
↪stores actual vertices and path_index is current index in path[]'''

  def getAllPathsUtil(self, u, d, visited, path):
    # 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:
      #print(path)
      self.all_paths.append(path.copy())
    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:
          self.getAllPathsUtil(i, d, visited, path)

    # Remove current vertex from path[] and mark it as unvisited
    path.pop()
    visited[u]= False


    # Prints all paths from 's' to 'd'
  def getAllPaths(self, s, d):

        # 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.getAllPathsUtil(s, d, visited, path)


  def get_all_path(self,s, d):
    self.all_paths = []
```

```
    self.getAllPaths(s, d)
    return self.all_paths
```

Nodes_dict is a dictionary which contains informations about which are cities are interlinked and what is distance between them.

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

graph = Graph(len(Nodes_dict.keys()))
for node_1, node_2,_ in args.nodes:
  graph.addEdge(int(node_1), int(node_2))
  graph.addEdge(int(node_2), int(node_1))
```

Car Class is representation of each car. This class is used to store all the details of car like current_location,path travelled, tr , source ,destination,init_battery,Max_battery,charging_rate,discharging_rate, Average speed

```
[ ]: 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
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

def check_car(all_cars,current_car):  #Function to check if there is any other␣
 ↪car at same node as the given 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 all_paths_combinations(cars,graph):  #Function to get all possible␣
 ↪combinations of all possible paths of each car
  list_ = []
  for car in cars:
    list_.append(graph.get_all_path(int(car.source), int(car.destination)))
  all_paths_combinations = [p for p in itertools.product(*list_)]
  return all_paths_combinations
```

This is implemtation of Depth First Search to find path with minimum of max(tr). When the algorithm is runned more than once it may provide results i.e it may produce different path then from previous run but the algorithm is successful each time in the achieving the objective i.e. minimum of max(tr).

The different results is due to reason that inital a path is choosen randomly from all possible combinations of all possible paths of each car which can be different. Since the objective of algorithm is minimize max(tr) and minimize tr of each other the results obtained on each run of algorithm can be different.

How the code works=>

1) In the below code we first initalize the best_path_time = infinity
2) All possible sets of paths which leads each car from their source to destination is are calculated using function all_paths_combinations.
3) Then choose a random set of paths from all possible sets.
4) The cars follow the their repsective path and reach their destination.
5) The tr of each car is notted. If max(tr) is less than max(tr) of best_path_time then the best_path_time is updated to this current path time and best_path is updated to the current set of paths.
6) Step 3-5 are repeated untill all sets are explored

[ ]:
```
"""##DFS""" # Depth First Search To find path with minimum of max(tr)
if not args.Pruning:
    cars = get_cars(args)
    all_paths = all_paths_combinations(cars,graph)
    best_path = []
    best_path_time = [np.inf]*len(cars)
    while all_paths!=[]:
        paths_choosen_index  = (np.random.choice(len(all_paths),1)).reshape(())
        paths_choosen =  list(all_paths[paths_choosen_index])
        path_time = []
        cars = get_cars(args)
        while cars!=[]:
            for i,car in enumerate(cars):
```

```
                current_node  = car.current_location
                if int(car.current_location)==int(car.destination):
                    continue
                childs,childs_dis = Nodes_dict[current_node]
                current_node_index = paths_choosen[i].index(int(current_node))
                next_node = paths_choosen[i][current_node_index+1]
                choosen_child_index = childs.index(int(next_node))
                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,car)
            for j,car in enumerate(cars):
                if car.current_location==car.destination:
                    path_time.append(car.tr)
                    cars.remove(car)
                    paths_choosen.pop(j)
        if path_time!=[]:
            if max(path_time) < max(best_path_time):
                best_path = list(all_paths[paths_choosen_index])
                best_path_time = path_time

        all_paths.pop(paths_choosen_index)
    print('By DFS Method:')
    print('Best path: ', best_path)
    print('Best path Time: ', best_path_time)
    print('Max Tr: ', max(best_path_time))
```

This is implemtation of Depth First Search combined with pruning to find path with minimum of max(tr).

Here pruning happens when the tr of any car exceeds the minimum max(tr) found untill now by algorithm.

Here also one can obtain different path due to reasons stated previously

How the code works=>

1) In the below code we first initalize the best_path_time = infinity

2) All possible sets of paths which leads each car from their source to destination is are calculated using function all_paths_combinations.

3) Then we choose a random set of paths from all possible sets.

4) The cars follow the their repsective path and reach their destination. But while this happens if tr of any car exceeds the max(tr) of best_path_time then all the set is prunned and algorithms proceed with another set.

5) If all cars successfully each their destinations then the tr of each car is noted. If max(tr) is less than max(tr) of best_path_time then the best_path_time is updated to this current path time and best_path is updated to the current set of paths.

6) Step 3-5 are repeated untill all sets are explored

[ ]:
```
"""##Pruning""" # Depth First Search combined with pruning to find path with␣
↪minimum of max(tr)
if args.Pruning:
    cars = get_cars(args)
    all_paths = all_paths_combinations(cars,graph)
    best_path = []
    best_path_time = [np.inf]*len(cars)
    while all_paths!=[]:
        paths_choosen_index  = (np.random.choice(len(all_paths),1)).reshape(())
        paths_choosen =  list(all_paths[paths_choosen_index])
        path_time = []
        cars = get_cars(args)
        while cars!=[]:
            for i,car in enumerate(cars):
                current_node  = car.current_location
                if int(car.current_location)==int(car.destination):
                    continue
                if car.tr > max(best_path_time):
                    path_time = []
                    break
                childs,childs_dis = Nodes_dict[current_node]
                current_node_index = paths_choosen[i].index(int(current_node))
                next_node = paths_choosen[i][current_node_index+1]
                choosen_child_index = childs.index(int(next_node))
                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,car)
            if car.tr > max(best_path_time):
                break
            for j,car in enumerate(cars):
                if car.current_location==car.destination:
                    path_time.append(car.tr)
                    cars.remove(car)
                    paths_choosen.pop(j)
        if path_time!=[]:
            if max(path_time) < max(best_path_time):
```

```
            best_path = list(all_paths[paths_choosen_index])
            best_path_time = path_time

        all_paths.pop(paths_choosen_index)
    print('By Pruning Method:')
    print('Best path: ', best_path)
    print('Best path Time: ', best_path_time)
    print('Max Tr: ', max(best_path_time))
```