ARTIFICIAL INTELLIGENCE : FOUNDATIONS AND APPLICATIONS

ASSIGNMENT 1 on Problem 3 - Electric vehicles

Team Members :

➔ Meghashrita Das	-	19AG3FP09
➔ Raghav Sharma	-	19AG30024
➔ Barchha Dhawal Samir	-	19AG3FP13

The aim of this project is to figure out how to run a state-search algorithm when one of our axes is continuous. So far, we've used simple discrete states along lines of motion between points on a grid, or finite moves in a game, whenever we've applied an algorithm.

When a continuous element such as time is introduced, however, all of our preconceived algorithms fail because there is no way to produce an infinite number of states corresponding to each point in time.

User Manual:

In order to use the code on your custom data, you will have to alter the TEST DATA part in the ending section of the notebook. The specific details of the variables and the values to be stored in them is as follows:

e<xy> (eg: e12, e23): This specifies the edge weights between the respective cities.

B<x> (eg: B1, B2): The initial battery of the respective vehicle.

M<x> (eg: M1, M2): The maximum battery of the respective vehicle.

S<x> (eg: S1, S2): The average speed of the respective vehicle.

D<x> (eg:D1, D2): The discharge rate of the respective vehicle.

C<x> (eg:C1, C2): Charging rate at the charging station.

V<x> (eg:V1, V2): Cities defined with their number, charging rate and distance from other cities.

car<x> (eg:car1, car2): Cars defined with their source, destination, initial battery, discharge rate, maximum battery, and average speed respectively.

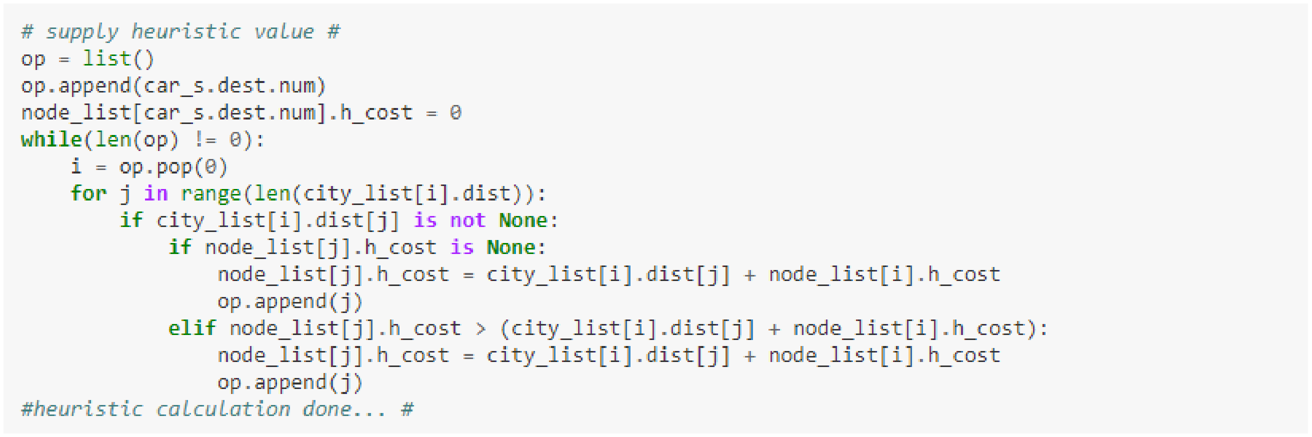
city_list: List of all cities defined above.

car_list: List of all cars defined above.

The code outputs the time taken by each vehicle in order to reach its destination. The maximum of these values is the total time taken for all vehicles to reach their destination.

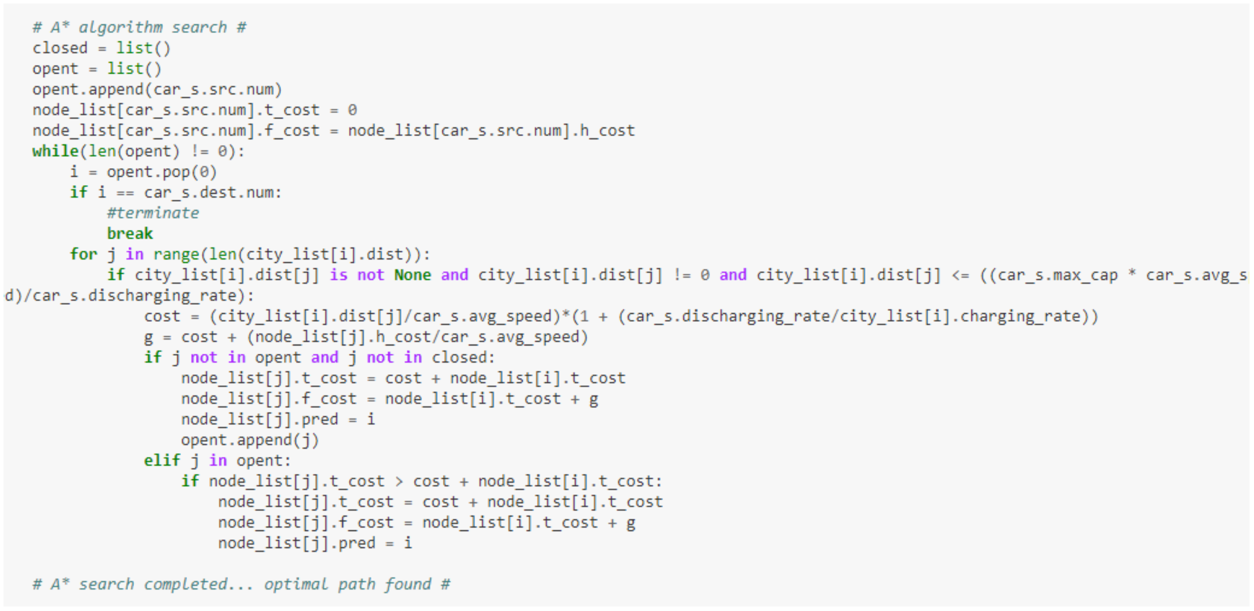
Algorithm :

1. First a function is used to collect the data about city and vehicle respectively
2. Then we are calculating Heuristics of distances in between each city



```
# supply heuristic value #
op = list()
op.append(car_s.dest.num)
node_list[car_s.dest.num].h_cost = 0
while(len(op) != 0):
    i = op.pop(0)
    for j in range(len(city_list[i].dist)):
        if city_list[i].dist[j] is not None:
            if node_list[j].h_cost is None:
                node_list[j].h_cost = city_list[i].dist[j] + node_list[i].h_cost
                op.append(j)
            elif node_list[j].h_cost > (city_list[i].dist[j] + node_list[i].h_cost):
                node_list[j].h_cost = city_list[i].dist[j] + node_list[i].h_cost
                op.append(j)
#heuristic calculation done... #
```

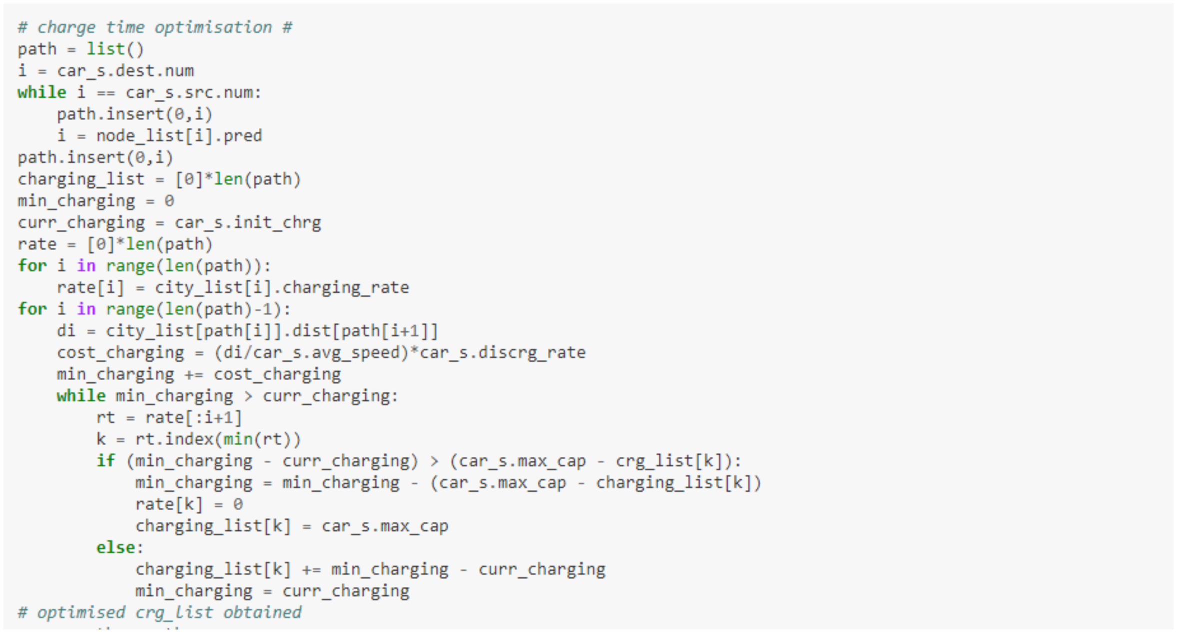
3. Now we apply A* algorithm to get the optimal path across the node



```
  # A* algorithm search #
  closed = list()
  opent = list()
  opent.append(car_s.src.num)
  node_list[car_s.src.num].t_cost = 0
  node_list[car_s.src.num].f_cost = node_list[car_s.src.num].h_cost
  while(len(opent) != 0):
      i = opent.pop(0)
      if i == car_s.dest.num:
          #terminate
          break
      for j in range(len(city_list[i].dist)):
          if city_list[i].dist[j] is not None and city_list[i].dist[j] != 0 and city_list[i].dist[j] <= ((car_s.max_cap * car_s.avg_s
d)/car_s.discharging_rate):
              cost = (city_list[i].dist[j]/car_s.avg_speed)*(1 + (car_s.discharging_rate/city_list[i].charging_rate))
              g = cost + (node_list[j].h_cost/car_s.avg_speed)
              if j not in opent and j not in closed:
                  node_list[j].t_cost = cost + node_list[i].t_cost
                  node_list[j].f_cost = node_list[i].t_cost + g
                  node_list[j].pred = i
                  opent.append(j)
              elif j in opent:
                  if node_list[j].t_cost > cost + node_list[i].t_cost:
                      node_list[j].t_cost = cost + node_list[i].t_cost
                      node_list[j].f_cost = node_list[i].t_cost + g
                      node_list[j].pred = i

  # A* search completed... optimal path found #
```

4. Optimising the charge left and time required to find optimal path



```python
# charge time optimisation #
path = list()
i = car_s.dest.num
while i == car_s.src.num:
    path.insert(0,i)
    i = node_list[i].pred
path.insert(0,i)
charging_list = [0]*len(path)
min_charging = 0
curr_charging = car_s.init_chrg
rate = [0]*len(path)
for i in range(len(path)):
    rate[i] = city_list[i].charging_rate
for i in range(len(path)-1):
    di = city_list[path[i]].dist[path[i+1]]
    cost_charging = (di/car_s.avg_speed)*car_s.discrg_rate
    min_charging += cost_charging
    while min_charging > curr_charging:
        rt = rate[:i+1]
        k = rt.index(min(rt))
        if (min_charging - curr_charging) > (car_s.max_cap - crg_list[k]):
            min_charging = min_charging - (car_s.max_cap - charging_list[k])
            rate[k] = 0
            charging_list[k] = car_s.max_cap
        else:
            charging_list[k] += min_charging - curr_charging
            min_charging = curr_charging
# optimised crg_list obtained
```

What is A* algorithm ?

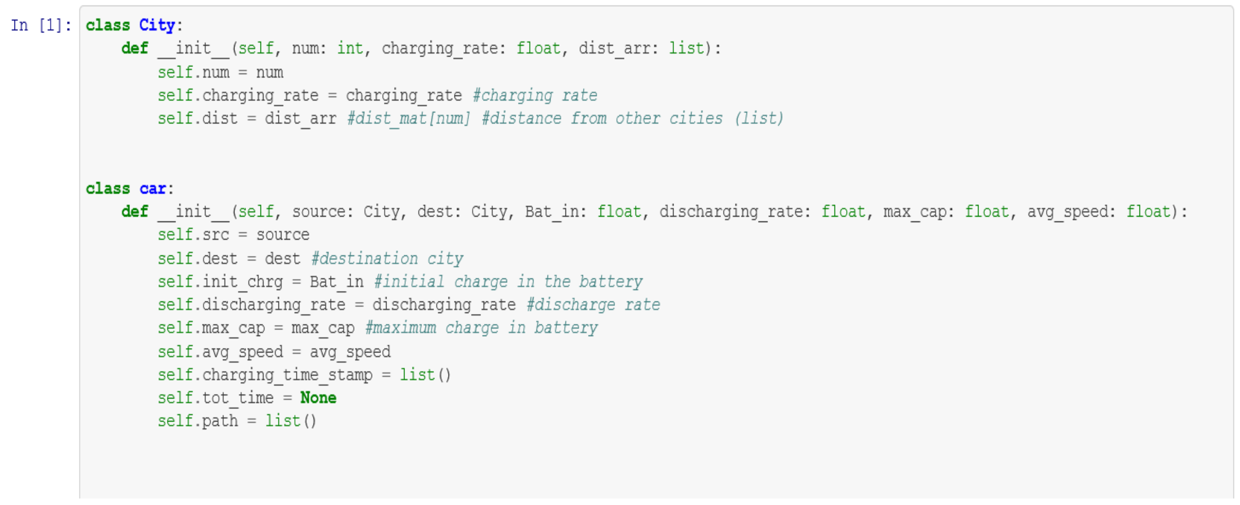
➔ A* Search algorithm is one of the best and popular techniques used in path-finding and graph traversals.

Why A* algorithm ?

➔ Informally speaking, A* Search algorithms, unlike other traversal techniques, have “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms.

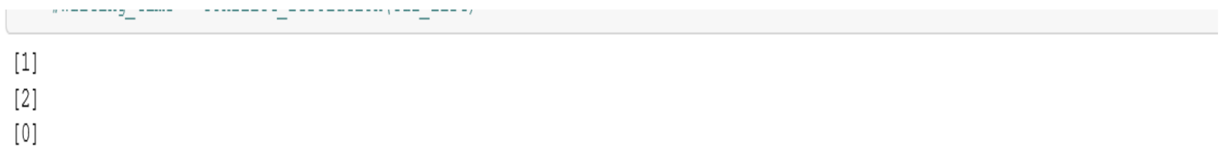
Output :

The following is the screenshot of the result we got on running the code on the sample data that we tested it on. The values are the time taken by the three vehicles we used in the sample data. It is clearly visible that the second vehicle takes the maximum time and is therefore the total time taken for all vehicles to reach their destination :



```
In [1]: class City:
            def __init__(self, num: int, charging_rate: float, dist_arr: list):
                self.num = num
                self.charging_rate = charging_rate #charging rate
                self.dist = dist_arr #dist_mat[num] #distance from other cities (list)


        class car:
            def __init__(self, source: City, dest: City, Bat_in: float, discharging_rate: float, max_cap: float, avg_speed: float):
                self.src = source
                self.dest = dest #destination city
                self.init_chrg = Bat_in #initial charge in the battery
                self.discharging_rate = discharging_rate #discharge rate
                self.max_cap = max_cap #maximum charge in battery
                self.avg_speed = avg_speed
                self.charging_time_stamp = list()
                self.tot_time = None
                self.path = list()
```



```
[illegible]
[1]
[2]
[0]
```