Fuel Station Management System

by

1. Deepak kumar - 205002024
2. Gokulakrishnan B-205002032

Problem Statement

Given n Fuel stations, m customers, and traffic in each petrol bunk, output the traffic in each fuel station by computing the nearest fuel station from the customer's location and recommend location to open a new petrol bunk using various algorithms and various data structures and calculating their time complexities.

Nature of data

- Inputs:
  - x and y coordinate of the fuel stations and customers.
- Output:
  - Grid representing customers and stations and the path between them.
  - New location (x and y coordinates) to open a new station.

Data Structures used to compute

i) Nearest neighbor

- K dim trees
- List (brute force algorithm)

ii) New x and y coordinates to open a station

- 2-dim grid
- Dictionary

Comparison of nearest neighbor algorithms

Brute force-manually compute distance between customer and every fuel station and choose the minimum one.

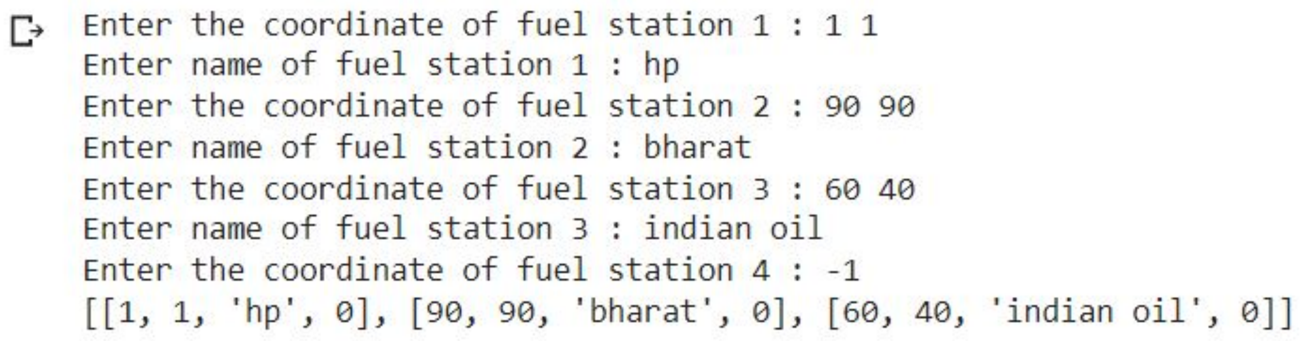
Optimized Algorithm-Using K dimensional trees to compute the nearest neighbor.

Analysis

Two modes of comparison:

- Wall time (execution time)
- No of times the for loop is executed

Compute 1 lakh random inputs for customers with 3 fuel stations.

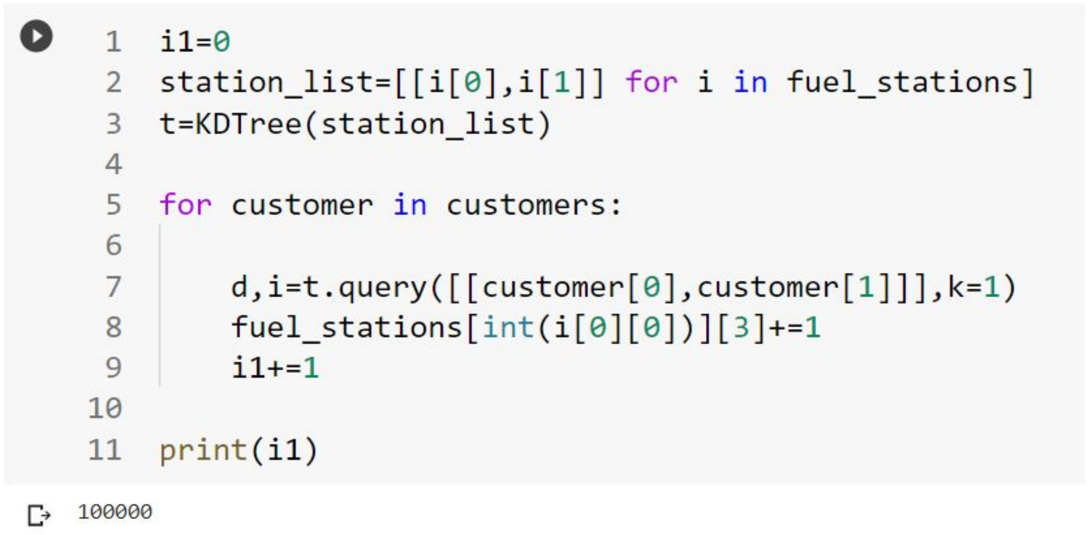


```
Enter the coordinate of fuel station 1 : 1 1
Enter name of fuel station 1 : hp
Enter the coordinate of fuel station 2 : 90 90
Enter name of fuel station 2 : bharat
Enter the coordinate of fuel station 3 : 60 40
Enter name of fuel station 3 : indian oil
Enter the coordinate of fuel station 4 : -1
[[1, 1, 'hp', 0], [90, 90, 'bharat', 0], [60, 40, 'indian oil', 0]]
```

Number of iterations

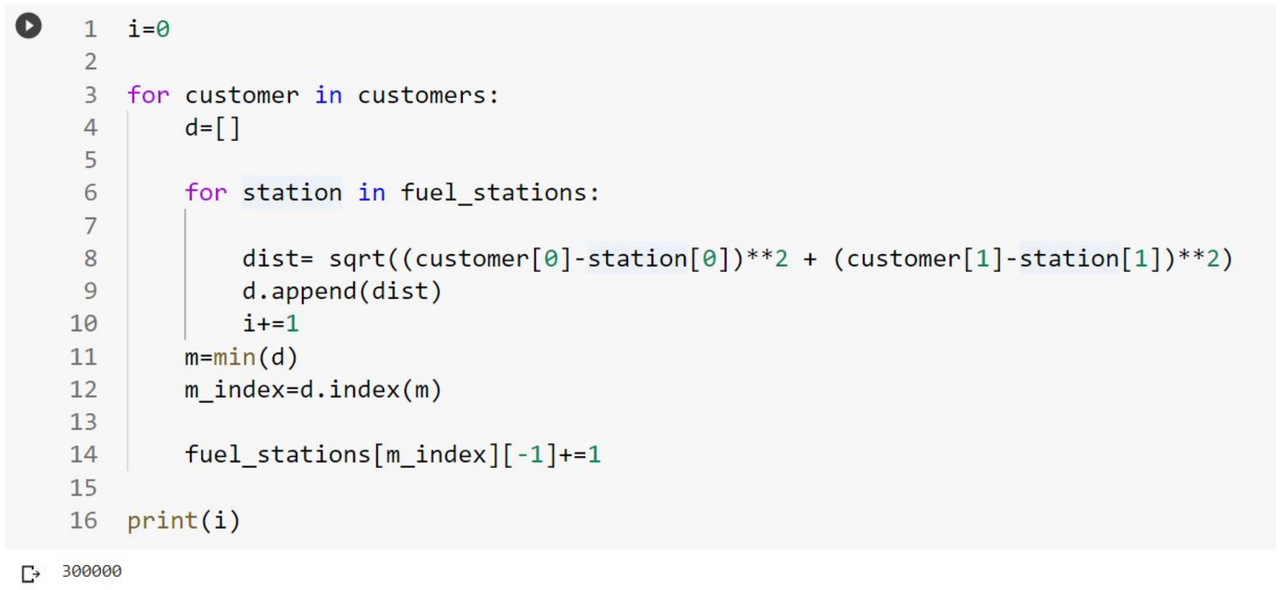
1.K dim trees

O(nlogn)



```
i1=0
station_list=[[i[0],i[1]] for i in fuel_stations]
t=KDTree(station_list)

for customer in customers:

    d,i=t.query([[customer[0],customer[1]]],k=1)
    fuel_stations[int(i[0][0])][3]+=1
    i1+=1

print(i1)
```

```
100000
```



```
i=0

for customer in customers:
    d=[]

    for station in fuel_stations:

        dist= sqrt((customer[0]-station[0])**2 + (customer[1]-station[1])**2)
        d.append(dist)
        i+=1
    m=min(d)
    m_index=d.index(m)

    fuel_stations[m_index][-1]+=1

print(i)
```

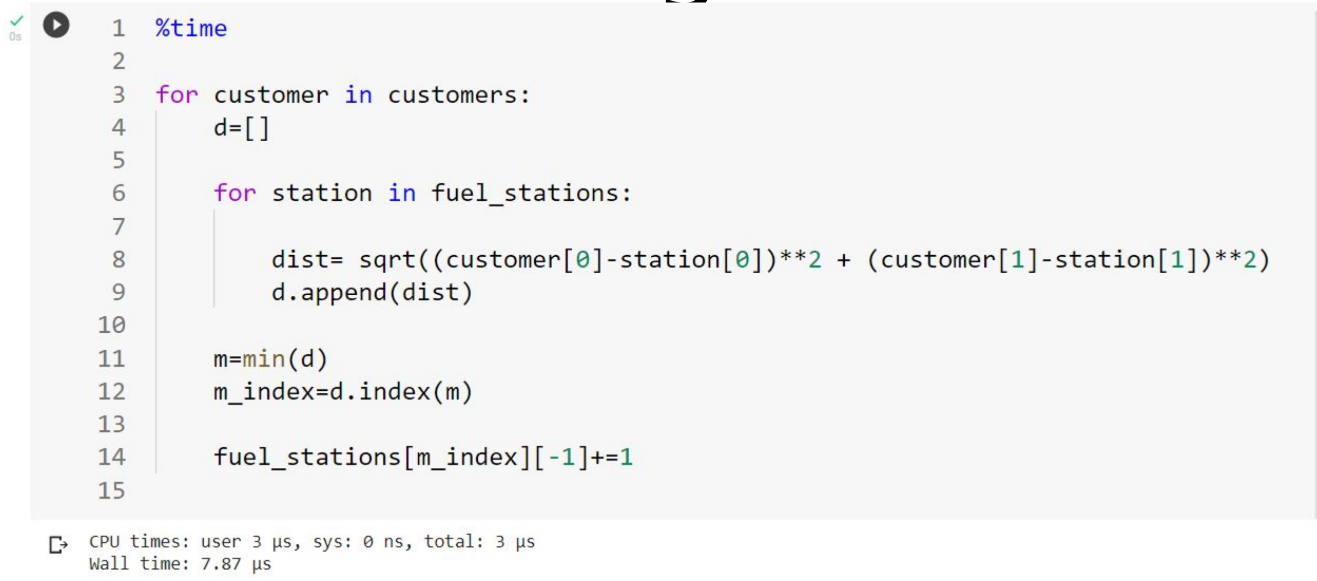
```
300000
```

2.Brute force

O(n.m)

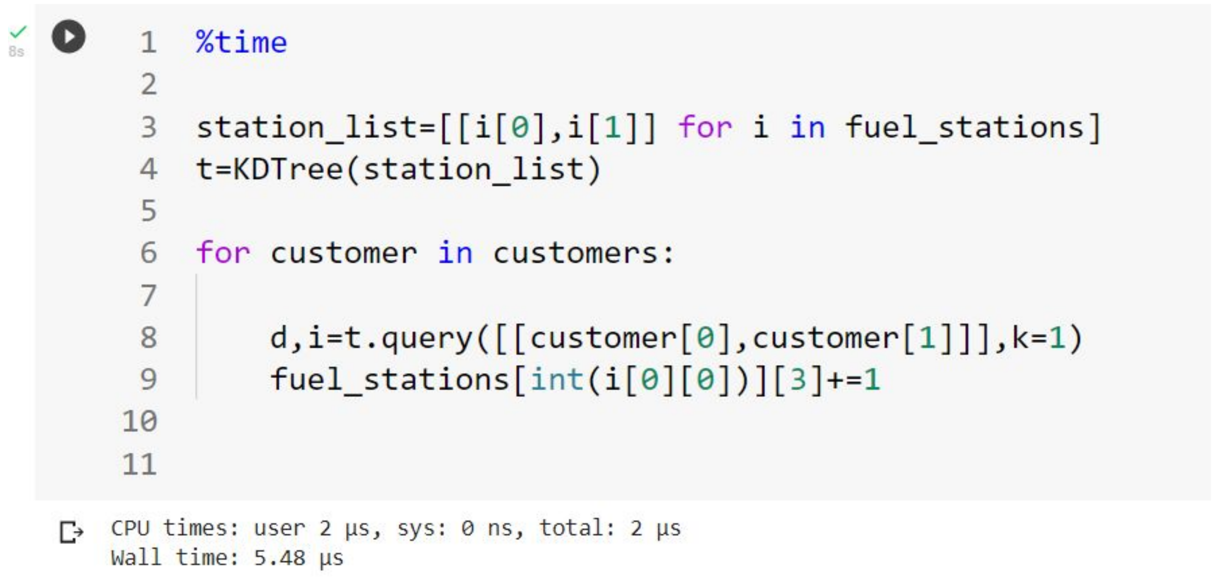
Execution time using %time

Brute force



```
%time

for customer in customers:
    d=[]

    for station in fuel_stations:

        dist= sqrt((customer[0]-station[0])**2 + (customer[1]-station[1])**2)
        d.append(dist)

    m=min(d)
    m_index=d.index(m)

    fuel_stations[m_index][-1]+=1

```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 7.87 µs
```



```
%time

station_list=[[i[0],i[1]] for i in fuel_stations]
t=KDTree(station_list)

for customer in customers:

    d,i=t.query([[customer[0],customer[1]]],k=1)
    fuel_stations[int(i[0][0])][3]+=1


```

```
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 5.48 µs
```

K dim trees

Analysis to recommend new station

We are dividing the main grid into smaller grids.

We use dictionary to compute the traffic in each grid.

Based on that we can recommend to open a station in the smaller grid with more traffic.

Analysis to recommend new station

If there are n customers, then we need n iterations to assign them to the grid.

Then we need n iterations to compute grid with maximum traffic.

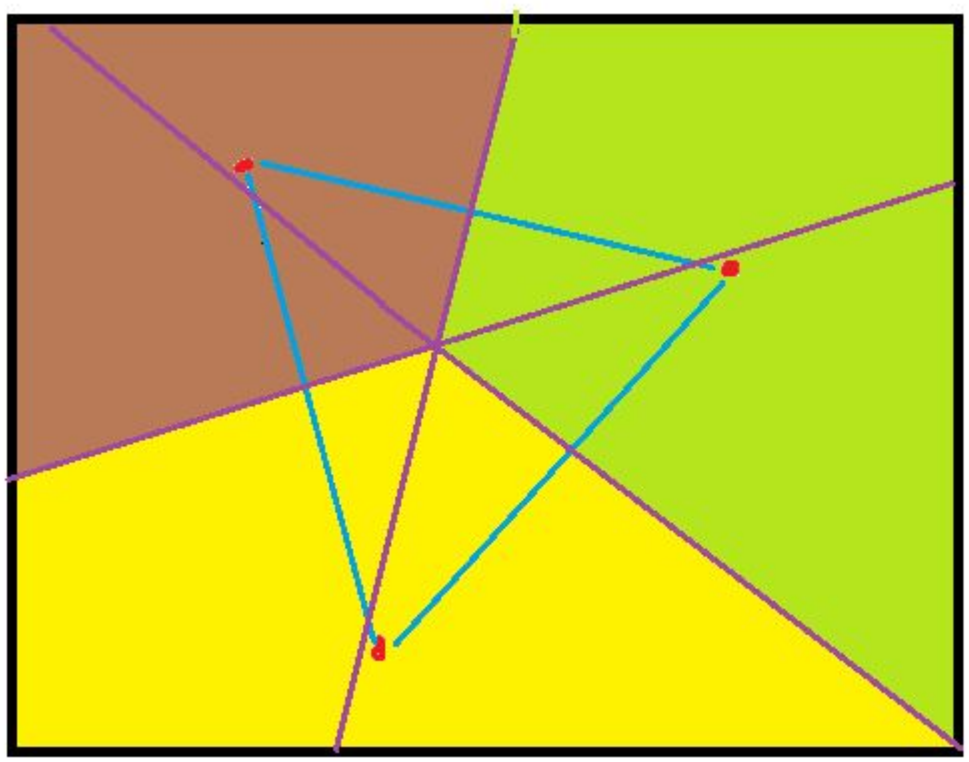
So time complexity will be O(n).

Gap Analysis

The efficient way of doing it in O(1) complexity for a given customer can be achieved by pre-computing the regions inside grid for each fuel station, by joining the points, compute the mid-point and draw the perpendicular line to the joining line at the mid-point and compute the region accordingly.

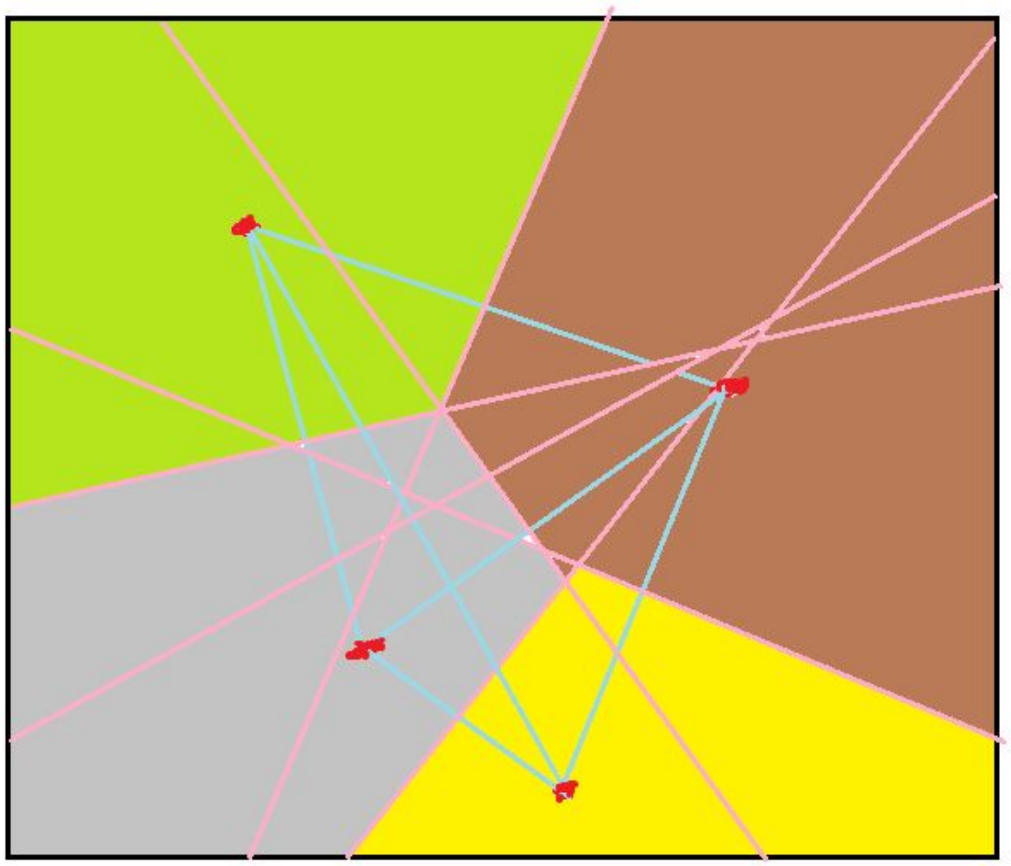
Gap Analysis

Illustration for 3 stations:



Gap Analysis

Illustration for 4 stations



Thank You