Introduction

In this challenge, I focused on the specific data structure to tackle that problem. Since our data is a kind of spatial data and we will subsequently do a lots of quests respectively to find the matching coordinates, in that sense, Kd-tree[1] data structure will be sufficient to reduce time complexity issue. Yes, Of course, there are another approaches that can be applied to solve that problem. For instance, one of them is k nearest neighbor searching (k-nn)[2]. However, the rest of them including k-nn have worse time complexity than Kd-tree. Kd-tree's complexity is O(log n) in terms of searching according to the table, below.

Algorithm	Average	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

So, I just skipped the rest due to those reasons. In this solution, we will focus on using the Kd-tree. A Python library was used to solve the problem quickly and efficiently. Therefore, we preferred to use C version Kd-tree implementation[3] in Scipy instead of using pure python implementation because of the reason, below.

- Its performance (cKDtree) is better than pure python implementation, KDTree.
- The API also is providing concurrency approach to scale up.

In addition, Kd-tree is really handy data structure to use in data mining and image processing. For instance, data mining algorithm, DBSCAN[4] is referring to Kd-tree to reduce time complexity in terms of searching. Actually,I came across this data structure around 2008 in my first data mining class. I used that data structure in the implementation of DBSCAN algorithm.

Solution

If we talk about our data in this challenge, let's say the input data has a M rows. The open traveling data has N rows. If we try to find the closest matching point by using brute-force approach, our time complexity will be $O(N^2M)$ [creating difference pairs N * M for each point, and multiply N times searching to the find closest points]

If we solved that problem using k-d tree, at the end of day, it takes O(N) to build a Kd-tree. After creating tree once, the query time will be O(M log M) for whole items in the input file. The total cost will be O(N) + O(M log M). O(N) can be ignored since we just take care about the searching time. Therefore, Kd-Tree's performance **O(M log M)** obviously will be better in average case, even worse case $O(M^2)$, too.

$$O(M \log M) < O(M^2) < O(N^2M) = O(M^3) \ [\text{if } N == M]$$

Usage Script

```
python solution.py -i sample_data.csv -o result.csv -d 0.1 -j 10
```

-i input file name
-o output file name
-d max distance in degree (this difference may be determined according to the demand. I guess, 1-2 km radius is fine in degree)
-j Number of jobs to schedule for parallel processing. If -1 is given all processors are used. Default: 1.

1 Kd-tree, https://en.wikipedia.org/wiki/K-d_tree
2 K-NN Searching, https://en.wikipedia.org/wiki/Nearest_neighbor_search
3 cKDTree, https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html
4 DBSCAN, https://en.wikipedia.org/wiki/DBSCAN