The Adaptability of Spatial Algorithms with Spatial-Temporal Trajectory Data

Longpeng Xu
MDataSci Candidate
The University of Queensland
Brisbane, Australia

***Abstract*—The research aims to compare the adaptability of multiple spatial algorithms in manipulating a spatial-temporal taxi trajectory dataset. In the research, multiple spatial indexing algorithms are used. Plus, algorithms for computing trajectory similarity are also concerned. The author found the mentioned algorithms outperform linear scan in terms of costs while not performance. The adaptability is also confined by syntax congruence between different languages.**

***Index Terms*—adaptability, spatial indexing, taxi trajectory, trajectory similarity, syntax congruence**

I. Introduction

Trajectory data has been a research hotspot, not only as it differentiates the efficiency of various indexing algorithms in computer science, but as it can be incorporated into public and private sectors, where researchers expect insights into urban planning, business optimization and consumer psychology, etc. distilled from data. Handling trajectory datasets, issues can occur: They are laborious to be preprocessed, as they are mostly high-dimensional, accompanied by spatial and other attributes. The adaptability of different algorithms to such datasets is not clear. For example, a trajectory of longitude-latitude pairs requires great-circle distance, which, however, is not supported by kd tree but by ball tree. By resolving some issues and for realizing algorithm comparison in terms of adaptability, the research gives an example in manipulating and applying algorithms on a taxi trajectory dataset.

In the context of the research, 'adaptability' covers the performance and the costs of algorithms, and syntax congruence, as the example illustrates. The research assumes that the great-circle distance is the optimal method to calculate distances, despite that the earth is not perfectly spherical. In extra, it is assumed that for the same trajectory, the start time overwrites the actual time of passing each point (longitude-latitude pair).

The remaining parts of the article are Dataset, Methodology, and Results.

II. Dataset

The publicly available dataset by Cross [**?**] collects, almost completely, the trajectories of 442 taxis in a Portuguese city, spanning from 1 July 2013 to 30 June 2014. The dataset is relational as column `TRIP_ID` is unique, with around 1.71 million rows and 9 columns in a CSV file of 1.94 gigabytes. Columns `CALL_TYPE` and `ORIGIN_STAND` indicates whether a trajectory starts from a taxi stand and the ID of the stand. `TIMESTAMP` is provided and `POLYLINE` is the trajectory array of shape $n \cdot 2$ appeared as a string, which brings remarkable challenges to preprocessing and analytics.

III. Methodology

The research implements R-tree, ball tree, kd tree, Hausdorff distance, and DTW distance, based on different queries. In order to compare the time costs and the memory costs in Results, a linear scan algorithm is also developed for each query. To compute the precision, the recall and the f1 score of each case, ground truth results are generated using PostgreSQL for each query.

A. Retrieve the points in a specified rectangular area and within a given time window

The R-tree algorithm is a perfect candidate for rectangular query, since the Python library of R-tree has the left, right, bottom and top parameters instantiate a rectangular. Algorithm **??** implements the query. Moreover, in terms of the code, the linear scan algorithm for the query is even simpler, just applying for-loop and if-suite and iterating through each point.

Algorithm 1 R-tree for rectangular query

```
function RTREE_AREA_TIME(rect, time, points)
    input: the rectangle, the time window and data
    output: the points confined by the inputs
    create idx, an R-tree index
    for i in points do
        Insert [pointid, lon, lat] to idx
    end for
    Intersect rect with idx yields in_area, the list of
    points within the rectangle
    return sorted list of points filtered by time
end function
```

B. Access k nearest neighbours (i.e., points) of a specified trajectory

The basic idea of k nearest neighbours is of a trajectory is to find k points that have the minimum distances to the trajectory. Further, since a trajectory consists of multiple points, the distance from a point to a trajectory is defined by the minimum value among the distances from the point to every trajectory point. Concisely, in this research, k nearest neighbours of a trajectory are defined by,
if $k = 1$,

$$\min_{p_i \in U - tr} \left(\min_{p_j \in tr} \text{dist}(p_i, p_j) \right), \tag{1}$$

if $k \geqslant 1$, for $p_i \in U - tr$,

$$\left(\min_{p_j \in tr} \text{dist}(p_i, p_j) \right)_{(1)}, ..., \left(\min_{p_j \in tr} \text{dist}(p_i, p_j) \right)_{(k)} \tag{2}$$

where
tr is the specified trajectory,
p_j is any point on tr,
U is the universal set of the trajectories including tr,
p_i is any point not on tr (so that $p_i \in U - tr$), and
$(\cdot)_{(1)}, ..., (\cdot)_{(k)}$ are the k smallest elements of sequence $(\cdot)$.

The k nearest neighbours are defined in this manner mainly for the following reasons:

(1) This definition ensures the uniqueness of the k nearest neighbours. This is because a point outside a trajectory can be the nearest neighbour of two or more trajectory points at the same time, while computing $\min_{p_j \in tr} \text{dist}(p_i, p_j)$ guarantees the one-to-one mapping between a point and its distance to the trajectory.
(2) A self-built algorithm bears higher adaptability, as the intricate differences among existing libraries are avoided and the syntax congruence between Python and PostgreSQL is somehow guaranteed.

The query is implemented by a ball tree, a variant of the binary tree where a node is a ball or a hypersphere, as in Figure 1 by Zhu et al. [**?**]. The balls (nodes) may intersect each other, but this does not affect that a point belongs to only one ball, depending on its shortest distance to a spherical centre. For its spherical structure, a ball tree is highly capable of high-dimensional data, as stated by Wan and Lee [**?**]. An attractive property is that it supports great-circle distance. The implementation of a ball tree is shown in Algorithm **??**. Note that in step 5, the distances from each of the points to the trajectory are calculated all at once. The distances are consistent with the definition by parameter setting.

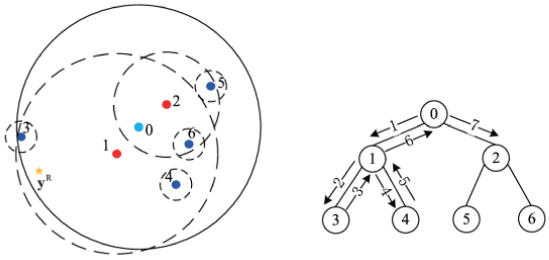

Fig. 1. An example of ball tree.

Algorithm 2 Ball tree for $k$nn query

1: **function** KNN_TRIPID(k, *tripid*)
   input: the number of nearest neighbours k and the given trajectory *tripid*
   output: pointid paired with distance of k nearest neighbours, sorted from the nearest
2: create *tripid_rows* storing indices of the trajectory points accessed by *tripid*
3: create *balltree* on the trajectory points
4: create *query_points_array* storing the points outside the trajectory
5: compute the *distances* from each point in *query_points_array* to the trajectory *tripid*, by *ball_tree*
6: sort $k$nn by the *distances* in *res*
7: convert the distances in degrees to meters equivalent in *res*
8: **return** *res*
9: **end function**

The most remarkable difference between linear scan and ball tree implementations also lies in step 5, where linear scan requires nested for-loops to achieve the same.

C. Retrieve the points within a certain distance to a specified trajectory

The Ball tree implementation of this query calls the function `knn_tripid` in Algorithm **??**, which makes it efficient. As in Algorithm **??**, it applies an if-elif suite and recursion. In essence, it returns the k nearest neighbours such that, the distance from the kth neighbour to the trajectory is smaller than *dist* while $(k + 1)$th is greater than *dist*.

D. Access the points in a circle area centred at a given taxi stand

To improve the diversity of algorithms in the research, the query is implemented by kd tree. The realisation is relatively quick, as in Algorithm **??**. The linear scan implementation loses the sense of circle query, as it merely calculates the distance from every point to the taxi stand and filters the points by a distance, the radius of the circle.

Algorithm 3 Within-distance-to-trajectory query based on $k$nn on Ball tree

function WITHIN_DIST($tripid, dist, k = 10, \Delta_k = 5$)
input: the trajectory *tripid*, the distance *dist*
output: pointid paired with distance of the points within *dist* from trajectory *tripid*
  create *knn_res_list* storing the output of `knn_tripid`$(k, tripid)$
  if max distance in *knn_res_list* $\geqslant$ *dist* **then**
    while max distance in *knn_res_list* $>$ *dist* **do**
      delete max distance and the paired pointid
    end while
  else if max distance in *knn_res_list* $\geqslant$ *dist* **then**
    knn_res_list = `within_dist`$(tripid, dist, k + \Delta_k)$
  end if
  return *knn_res_list*
end function

Algorithm 4 kd tree for circle query

function IDS_IN_CIRC($pointid, dist$)
input: the *pointid* of a taxi stand, the distance *dist*
output: the points circled within *dist* from taxi stand *pointid*
  create kd tree on data
  create *point* storing (longitude, latitude) of *pointid*
  query kd tree on *point* and converted *dist*, from meter to degree, to get the *circled_points*
  return the sorted *circled_points*
end function

Conversion from meter to degree in Algorithm **??** and the other around in Algorithm **??** is worth mentioning. A great circle distance between two points is equivalent to the arc length on the great circle (having its edge pass through the two points and the centre coincides with the spherical centre). The arc length is proportional to the central angle formed by the spherical centre and the two points [**?**]. Hence, the conversions can be deduced with constant factors only.

From meter to degree:

$$\frac{meter}{\text{earth\_radius} \cdot 1000} \cdot \frac{180}{\pi} = degree \tag{3}$$

Firstly, $\frac{meter}{\text{earth\_radius} \cdot 1000}$ converts the arc length in meters to the radian of the angle of the arc, given the definition of a radian (The central angle where arc length equals radius). Then, the radian is converted to degrees by multiplying $\frac{180}{\pi}$, since 2π rad = 360 deg. Reversely, from degree to meter:

$$degree \cdot \frac{\pi}{180} \cdot \text{earth\_radius} \cdot 1000 = meter \tag{4}$$

Firstly, convert degree to radian using the identity. Then, convert the angle in radian α, to arc length in meter l by $l = \alpha R$ where R is the earth radius. The earth's radius is 6371 kilometers on average.

E. Retrieve k most similar trajectories of a queried trajectory

This task is implemented in the algorithms of Hausdorff distance, linear scan (based on the definition of Hausdorff distance), and Dynamic Time Warping (DTW) distance. Syntax congruence issue occurs: While these algorithms can be executed on Python, PostgreSQL only offers a non-standard version of Hausdorff distance and cannot support DTW distance.

According to Rote [**?**], the Hausdorff distance between two trajectories, by modifying Equation (**??**), is defined as

$$\max_{p_i \in tr_1} \left(\min_{p_j \in tr} \text{dist}(p_i, p_j) \right) \tag{5}$$

where tr_1 is another trajectory and p_i is any point in the tr_1. We still follow the definition of 'the distance from a point to a trajectory' in Part B, assuming that 'a point' belongs to tr_1. The Hausdorff distance between the trajectories is the maximum of such kind of distances. Further, to retrieve k most similar trajectories of a queried trajectory, we retrieve the k smallest Hausdorff distances. The pseudocode is in Algorithm **??**. Due to the page limit, the linear scan and the DTW distance implementations are not elaborated.

Algorithm 5 Hausdorff distance for trajectory similarity query

function HSDF_MOST_SIM_TR($tripid, k$)
input: the queried trajectory *tripid* and the number of similar trajectories k
output: the k most similar trajectories paired with Hausdorff distances
  create *other_tr* = [] storing the other trajectories
  create *other_tripid* = [] storing the other tr. ids
  create *dist* = [] storing the Hausdorff distances
  create *queried* storing the points of the queried trajectory *tripid*
  for i in the other tripids **do**
    tr_tripid storing the points of trajectory i
    other_tr += [*tr_tripid*]
    other_tripid += [i]
  end for
  for *tr* in *other_tr* **do**
    dist += [Hausdorff distance between *queried* and *tr*]
  end for
  create *tripid_dist* by zipping *other_tripid*, *dist*
  return the k most similar tr. from *tripid_dist*
end function

IV. Results

The results are shown in Figure 2, corresponding to the five query tasks in Methodology. For each task, there

are four cases; for each case, the two algorithms (linear scan and the competing algorithm, for example, R-tree, in Task A) are tested. The left-aligned y-axis corresponds to performance measures, while the right-aligned y-axis corresponds to cost measures.

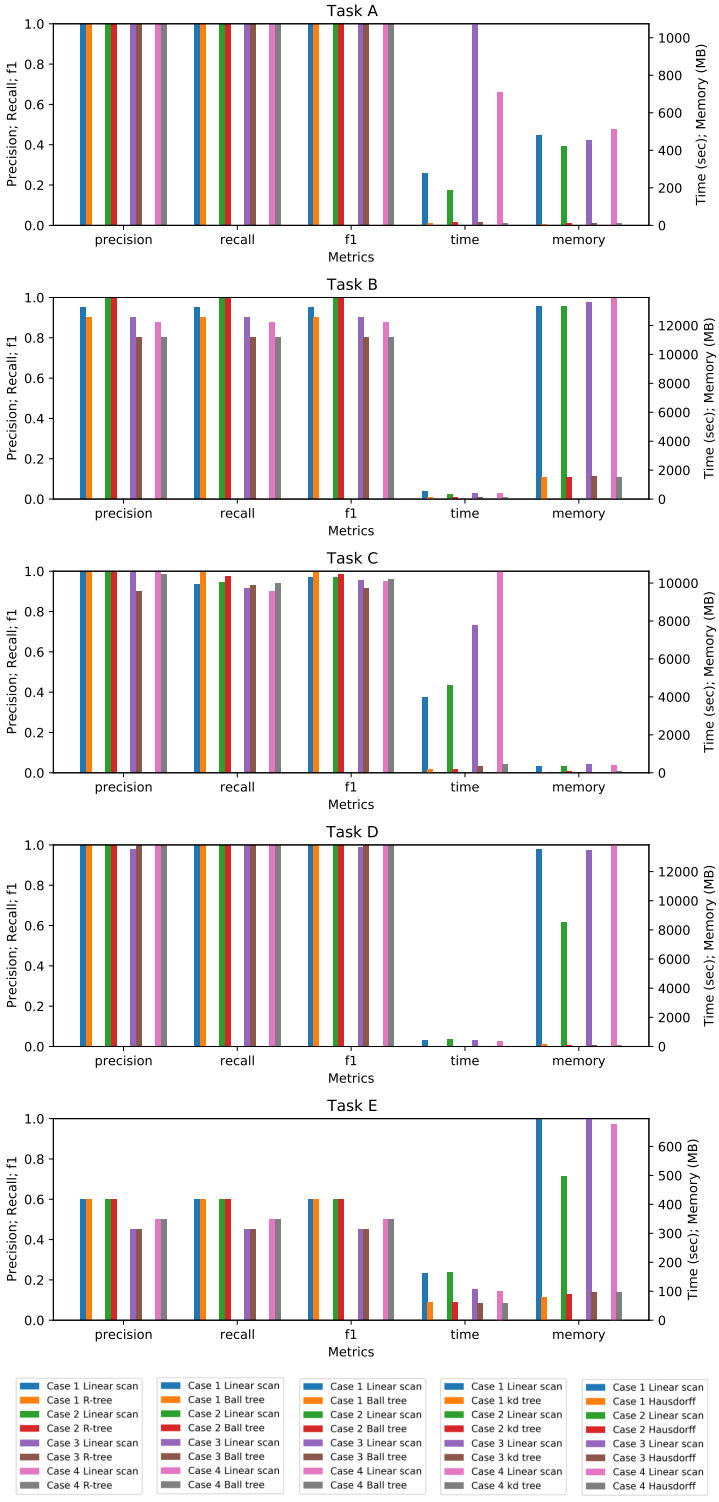

Fig. 2. Experiment results.

The main findings of the research, with comments, are:

(1) In terms of the performance measures (precision, recall, f1), overall, there is no striking difference between linear scan and the competing algorithm. However, this is not because the competing algorithms are incapable but in most cases, linear scans achieve very high performance at the cost of time and memory.

(2) In terms of the costs (time cost and memory cost), the competing algorithms aggressively outperform linear scans most of the time. Hence, the competing algorithms bear higher adaptability, even though not in the full spectrum as defined in the beginning.

(3) The syntax congruence affects the adaptability of algorithms observably. Beside Task E where there is no equivalent Hausdorff distance algorithm in PostgreSQL, the higher performance of linear scan in Task 2 also reflects the effect of syntax congruence: Only the equivalent of linear scan but not ball tree can be implemented in PostgreSQL. This indicates that optimizing the ground truth results is also significant to evaluate the performance of the algorithms.

(4) In the context of spherical-coordinate data, the adaptability of a spatial algorithm can be improved by the accurate unit conversion of distance. The example is Task D where kd tree does not support great-circle distance, while the ground truth by PostgreSQL is based on a World Geodetic System. Nevertheless, with accurate conversion, the full-mark performance by the kd tree is much beyond expected.

(5) The simpler queries are more likely to achieve full-mark performance across different algorithms and different cases. Remarkable examples include Task A (rectangular query) and Task D (circle query).

References

[1] C. Cross. Taxi Trajectory Data, *www.kaggle.com*. https://www.kaggle.com/datasets/crailtap/taxi-trajectory

[2] J. Zhu, E. Yu, Q. Li, H. Chen, and S. Shamai Shitz, Ball-Tree-Based Signal Detection for LMA MIMO Systems, IEEE communications letters, vol. 26, no. 3, pp. 602606, 2022, doi: 10.1109/LCOMM.2021.3140094.

[3] W. Wan and H. J. Lee, Deep feature representation and ball-tree for face sketch recognition, *International journal of system assurance engineering and management*, vol. 11, no. 4, pp. 818823, 2020, doi: 10.1007/s13198-019-00882-x.

[4] Great circle, Wikipedia, May 07, 2020. https://en.wikipedia.org/wiki/Great_circle

[5] G. Rote, Computing the minimum Hausdorff distance between two point sets on a line under translation, *Information processing letters*, vol. 38, no. 3, pp. 123127, 1991, doi: 10.1016/0020-0190(91)90233-8.