User Oriented Trajectory Similarity Search

Haibo Wang
School of Information Technology & Electrical Engineering, The University of Queensland
uqhwan15@uq.edu.au

Kuien Liu
The Institute of Software
Chinese Academy of Sciences
kuien@iscas.ac.cn

ABSTRACT

Trajectory similarity search studies the problem of finding a trajectory from the database such the found trajectory most similar to the query trajectory. Past research mainly focused on two aspects: shape similarity search and semantic similarity search, leaving personalized similarity search untouched. In this paper, we propose a new query which takes user's preference into consideration to provide personalized searching. We define a new data model for this query and identify the efficiency issue as the key challenge: given a user specified trajectory, how to efficiently retrieve the most similar trajectory from the database. By taking advantage of the spatial localities, we develop a two-phase algorithm to tame this challenge. Two optimized strategies are also developed to speed up the query process. Both the theoretical analysis and the experiments demonstrate the high efficiency of the proposed method.

1. INTRODUCTION

Trajectory similarity search is a hot research topic in recent years due to its broad range of applications, such as friend recommendations, trip planning, traffic analysis and carpooling. It studies the problem of finding a trajectory from the trajectory database such that the found trajectory is most similar to the query trajectory. A fair amount of research works were involved in the past decades on this topic, some [1, 15, 11, 16, 9, 4, 3, 13] focused on shaped-based similarity search, in which each trajectory consists of a sequence of equally important sample points; while others [17, 10, 20, 19, 12, 14] focused on the semantic aspects, in which each trajectory is represented as a sequence of meaningful entities, such as POIs, locations or regions.

The similarity for the shape-based search depends on how many common parts the trajectories share, while for the semantic-based search it depends on how many *significant* common parts shared. Clearly, the second kind of similarity is more reasonable as it considers more of those significant parts instead of treating each part equally. To measure the second kind similarity, it employes variuos huristic methods to identify which parts are important and meaningful and this, inherently, has two disadvantages:

- Due to the quality of the training data or the imperfection of the mining method, it cannot gurantee to find all of the important parts. Some significant places may be missed.
- Among all those important parts, they are not equally important to all users. In other words, this method cannot provide personalized service to different users.

To overcome these disadvantages, we propose a new type of query: *user-oriented trajectory similarity search*, in which each user can specify the relative importance of each part in the query trajectory. This new query has the advantages that:

- It can provide personalized query without missing those important parts, as whose importance have been designated by the user.
- It supports flexible and even highly complex query patterns. For instance, the relative importance of all parts in the query trajectory follows standard normal distribution.

The key challenge here is the efficiency issue. More specifically, given a user specified query, how to efficiently find out the most similar trajectory from the trajectory database which may contain huge number of trajectories.

Unfortunately, no existing solutions can readily be used to conquer this challenge. Vlachos et al. in [13] explore discovering similar multidimensional trajectories by building a cluster-based hierarchical indexing tree. This method, however, suffers from finding good clusters and representing points to build the hierarchical indexing tree on which the performance heavily depends. Besides, it considers shape similarity allowing spatial shifting between trajectories, which is totally different from our settings: no spatial shifting is allowed. Chen et al. in [3] exploit edit distance to measure trajectory similarity and provides three pruning techniques to efficiently retrieve the most similar trajectory. But under the user specified query in which the relative importance of every sample points are considered, these pruning techniques are no longer hold. A typical example is that one pruning technique works by bounding the number of *common Q-grams* for two sequences within *edit distance* k, where k is supposed to be an integer, while in our settings, due to the relative importance involved, this condition is hard to be guaranteed.

In this work, our major contributions are:

1. We propose a new type of trajectory similarity search with the merits of considering users' preferences and supporting personalized query.

2. We carefully define a new data model for this query and develop an efficient method to answer the similarity query. Two optimized strategies are also developed to speed up the query process.

3. We do theoretical analysis, carry out experiments on real dataset and both demonstrate the high efficiency of our method.

The remainder of this paper is organized as follows. In section 2, we define the data model and the problem. We then develop a two-phase algorithm and two optimized strategies in Section 3. Theoretical analysis and the experiments are conducted in Section 4 and Section 5, respectively. We discuss related work in section 6 and conclude this paper in section 7.

2. DATA MODEL AND PROBLEM

With the advancement of the modern GPS technologies, it is not unreasonable to assume that all trajectories in the database have similar sampling rate. If not, we can interpolate the trajectory data to make them satisfied the assumption condition.

Let $T = \{p_1, p_2, \cdots, p_n\}$ be a data trajectory, where $|T|$ denotes the size of sample points in T. Each sample point $p_i = \langle lon_i, lat_i \rangle$ is a pair of real values, where lon_i and lat_i correspond to longitude and latitude respectively. A user trajectory database is a set of data trajectories $DB = \{T_1, T_2, \cdots, T_N\}$, where N is the number of trajectories in DB. Let $Q = \{q_1, q_2, \cdots, q_m\}$ be the query trajectory, in which each sample point q_i is a triple of real values, namely $\langle lon_i, lat_i, w_i \rangle$, where $w_i \geq 0$ is the user assigned weight indicating the importance of the sample point. We say sample points q_i and p_j *matched* if $|lon_i - lon_j| \leq \epsilon$ and $|lat_i - lat_j| \leq \epsilon$; here the ϵ is the matching threshold.

There are several methods to measure trajectory similarity: Euclidean-based methods [1, 15, 11], DTW [16, 9, 4], ERP [2], EDR [3] and LCSS [13]. Among these methods, the first three (Euclidean-based methods, DTW and ERP) are sensitive to noise; the EDR concerns more about the dissimilarities between trajectories, as it only penalizes the gaps while ignores their common parts; the LCSS, on the other hand, is more robust to noise and more accurate to compute the similar parts, because just like its name (Longest Common Subsequence), it measures how many common parts two trajectories share and a larger value implies a better similarity. Here in this paper we use the same idea behind LCSS to measure the similarity between a data trajectory and a query trajectory. Specifically, given the aforementioned T and Q, we define the similarity measure *Heaviest Common Subsequence (HCSS)* between them as the weighted sum of their longest common subsequence and denote it as $HCSS(T, Q)$. The specific value can be derived from the following recursive computation:

$$\begin{cases} 0 & \text{if } n = 0 \text{ or } m = 0, \\ w_1 + HCSS(Rest(T), Rest(Q)) & \text{if } p_1,\ q_1 \text{ are matched}, \\ max \begin{cases} HCSS(Rest(T), Q), \\ HCSS(T, Rest(Q)) \end{cases} & \text{otherwise} \end{cases}$$

where $Rest(.)$ denotes the rest part of a trajectory with the first sample point removed.

Problem. Given a query trajectory Q, find T from the trajectory database DB such that $HCSS(T, Q) > 0$ and

$$HCSS(T, Q) \geq HCSS(T', Q), \forall T' \in DB \text{ and } T' \neq T.$$

In the extreme case that $\forall T \in DB$, $HCSS(T, Q) = 0$, we would say no trajectory in the database is similar to the query trajectory and therefore have no obligation to return any trajectory.

3. QUERY PROCESSING

One naive solution for the problem is to compute the similarity values with every trajectory in the database, then choose the one with the largest $HCSS$ value. The cost of this method is prohibitively expensive simply because it has to load every data trajectory from the external memory into internal memory in order to compute the $HCSS$ value, which would introduce tons of IOs, not mention the computation cost is quadratic to trajectory length.

To reduce the IO as well as the computation cost, we have the following observation.

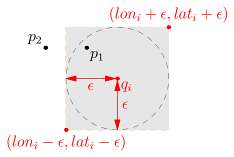
Observation 1 *Trajectories in the database are spatially scattered and the query trajectory is only within some limited area.*

Based on this observation, instead of retrieving all trajectories from the external memory, we only need to retrieve those trajectories having at least one sample point included by a small range that covers the query trajectory as a candidate set and it is highly likely that the candidate set contains the most similar trajectory. The key issue here is how large the *covering range* should be to make the candidate set surely containing the most similar trajectory. If it is too large, it may retrieve excessive number of trajectories at the price of a fair amount unnecessary IOs; on the other hand, if it is too small, it may miss the most similar trajectory as well. Therefore a proper covering range should be provided so that it dose not produce too many unnecessary IOs while at the same time it can still guarantee the correctness of the candidate set without introducing false dismissals.

Definition 1. ***(ϵ-buffer)*** Given a query trajectory Q, we define its *ϵ-buffer* $\mathcal{B}_\epsilon(Q)$ as the *union* of rectangular areas of all sample points, where the *rectangular area* of a sample point q_i is $[(lon_i - \epsilon, lat_i - \epsilon), (lon_i + \epsilon, lat_i + \epsilon)]$ and ϵ is the threshold (Figure 1).

THEOREM 1. *The ϵ-buffer of query trajectory Q can serve as a covering range without introducing false dismissals.*

PROOF. Let T be the most similar trajectory. From the problem definition we know $HCSS(T, Q) > 0$, which implies T must have at least one its sample point included by



$(lon_i+\epsilon, lat_i+\epsilon)$
p_2
p_1
q_i
ϵ
ϵ
$(lon_i-\epsilon, lat_i-\epsilon)$
(a)

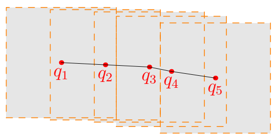


Figure 1: Examples of rectangular area and $\mathcal{B}_\epsilon(Q)$

the covering range $\mathcal{B}_\epsilon(Q)$ and therefore is included in the returned candidate set. □

Theorem 1 indicates us that we only need to search the query trajectory's ϵ-buffer to find the most similar trajectory, and in terms IO cost, it is a dramatic improvement over the naive method in which the correctness is achieved by searching the whole data space. Inspired by the above analysis, we get our first algorithm to find the most similar trajectory.

Before presenting the algorithm, we claim that every trajectory T in the database has a unique id called *tid*, and each sample point in T contains the *tid* information. If we get a sample point, we then know which trajectory it belongs to. Also we use an R-tree [7] to index all sample points in the trajectory database; by doing this, we can conduct range search on the R-tree and thus reduce the search cost.

Algorithm 1: *MST(TR, Q, ϵ)*

```
Input: R-tree root $TR$, query trajectory $Q$, threshold $\epsilon$
Output: The unique id of the most similar trajectory
1  $id \leftarrow \infty$;
2  /* Filter phase */
3  $List_\mathcal{C} \leftarrow \phi$; // the candidate set
4  foreach $q_i$ in $Q$ do
5      $rect \leftarrow [(lon_i-\epsilon, lat_i-\epsilon), (lon_i+\epsilon, lat_i+\epsilon)]$;
6      $List_\mathcal{P} \leftarrow TR.\text{rangeSearch}(rect)$;
7      foreach sample point $p$ in $List_\mathcal{P}$ do
8          $tid \leftarrow p.getTid()$;
9          if $List_\mathcal{C}.contain(tid)$ is not ture then
10             $List_\mathcal{C}.add(tid)$;
11 /* Refinement phase */
12 $distance \leftarrow -\infty$;
13 foreach $tid$ in $List_\mathcal{C}$ do
14     $T \leftarrow$ retrieve the $tid$th trajectory from external memory;
15     $hcss \leftarrow HCSS(T, Q)$;
16     if $distance < hcss$ then
17         $distance \leftarrow hcss$;
18         $id \leftarrow tid$;
19 return $id$;
```

The algorithm adopts a two phases strategy: filter and refinement. In the filter phase (lines 2-10), we generate a candidate set based on R-tree range search (i.e., search the area covered by $\mathcal{B}_\epsilon(Q)$) without introducing false dismissals; in the refinement phase (lines 11-19), we retrieve each trajectory appearing in the candidate set from the external memory, compute the exact similarity values, choose the one with the largest $HCSS$ value and return it as the most similar trajectory. The correctness of the algorithm is guaranteed by Theorem 1.

3.1 Adaptive Filter Strategy

If we look carefully at the filter phase, we will find that Algorithm 1 conducts range search for every sample point in the query trajectory. This, however, may not necessary, as indicated by the following observation.

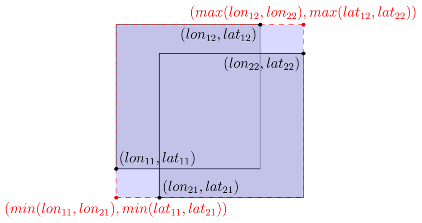
Observation 2 *The query trajectory consists of a series of sequential sample points which are usually spatially close.*

As a result, there are many overlaps between the range searches conducted in Algorithm 1. In Figure 1(b), we can clearly see the overlaps between consecutive range searches (rectangular areas). The multiple searching of these overlapping areas is obviously undesired as it will introduce unnecessary IO cost. To remedy this, we introduce the concept of *grouping consecutive query points*.

Definition 2. ***(Grouped area)*** For any two consecutive points q_i, q_{i+1} in the query trajectory, let their corresponding *rectangular areas* are $[(lon_{i,1}, lat_{i,1}), (lon_{i,2}, lat_{i,2})]$ and $[(lon_{i+1,1}, lat_{i+1,1}), (lon_{i+1,2}, lat_{i+1,2})]$, respectively. We group them together and define their *grouped area* $\mathcal{G}(q_i, q_{i+1})$ is as $[(lon_1, lat_1), (lon_2, lat_2)]$, where

$$\begin{aligned} lon_1 &= min(lon_{i,1}, lon_{i+1,1}) \\ lat_1 &= min(lat_{i,1}, lat_{i+1,1}) \\ lon_2 &= max(lon_{i,2}, lon_{i+1,2}) \\ lat_2 &= max(lat_{i,2}, lat_{i+1,2}) \end{aligned}$$

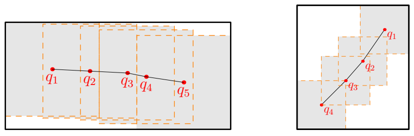
In the same way, we define grouped area for $k+1$ consecutive points q_i, q_{i+1}, $\cdots$, q_{i+k} as $\mathcal{G}(q_i, q_{i+k})$. Figure 2 shows an example, in which two smaller rectangular areas are grouped together to form the larger one with red dashed edges.



$(max(lon_{12}, lon_{22}), max(lat_{12}, lat_{22}))$
(lon_{12}, lat_{12})
(lon_{22}, lat_{22})
(lon_{11}, lat_{11})
(lon_{21}, lat_{21})
$(min(lon_{11}, lon_{21}), min(lat_{11}, lat_{21}))$

Figure 2: Grouped area $\mathcal{G}(q_1, q_2)$

To overcome the drawback caused by redundant IOs, one solution is to group certain number of consecutive points together so that we can have just one range search to cover the union of their rectangular areas. As illustrated in Figure 3(a), instead of conducting 5 range searches, we group the 5 query sample points together and only search once the grouped area $\mathcal{G}(q_1, q_5)$ bounded by the black box. This method works in the reason that it covers more than the original searching area (i.e., the *ϵ-buffer* of the query trajectory) and therefore guarantees no false dismissals, while the cost saved from overlapping search is *quite enough* to compensate the extra cost from searching new *dead space* (the white space between the shaded area and the black bounding box in Figure 3(a)).



(a) (b)

Figure 3: Points distribution vs. dead space

A problem with this solution is how many consecutive points shall be grouped together so that the extra cost introduced by searching new dead space will not cancel out the saved cost. If we group too few points, although the extra cost is small, it may not effective to reduce the redundant IO cost; if we group too many points, although the saved cost may be a lot, it may also introduce too much new dead space and the search cost on which would turn back canceling out the benefit earned from the saved cost. Therefore, in pursuit of keeping the total range search cost as small as possible, a proper grouping strategy shall be developed to ensure the balance between saving redundant IO cost and introducing new cost.

One strategy is simply grouping fixed number of consecutive points. This method, however, has the risk of bringing in too much dead space if the consecutive points are distributed as Figure 3(b) showing. To get rid of the risk, we need to take the grouping process under control and develop an *adaptive* grouping strategy. More specifically, we need to develop a *bound* for the search space so that after grouping, the grouped area will not surpass the boundary, thereby *bounding* the introduced dead space. Under this condition, we can aggressively group those consecutive points.

Definition 3. *(α-**boundary**)* Given a polyline $L = \{p_1, p_2, \cdots, p_n\}$ and $\alpha \geq 0$, let every line segment (p_i, p_{i+1}) be moved α distance along both directions that perpendicular to it to get the upper and lower lines, we define the **area** *between all upper and all lower lines* plussing *the two outer half circles centered at the end points of L with radius α* as the ***α-boundary*** of L, or $\mathcal{D}_\alpha(L)$, as illustrated in Figure 4.

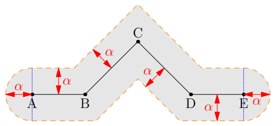


Figure 4: $L = \{A, B, C, D, E\}$ **and** $\mathcal{D}_\alpha(L)$

We can easily see that the *ϵ-buffer* of the query trajectory Q are covered by Q's *$\sqrt{2}\epsilon$-boundary*, namely $\mathcal{B}_\epsilon(Q) \subseteq \mathcal{D}_{\sqrt{2}\epsilon}(Q)$, because the perpendicular distance from every point in $\mathcal{B}_\epsilon(Q)$ to the polyline Q is less or equal than $\sqrt{2}\epsilon$. However, we cannot use this boundary as a bound to do grouping work owning to the fact that it is too tight to group any consecutive points. In other words, we need a more looser boundary.

Line simplification technique like Douglas-Peucker [6] algorithm may drop us a hint on this. Given a distance threshold $\delta > 0$ and a polyline specified by a sequence of n points $\{q_1, q_2, \cdots, q_n\}$, the goal of Douglas-Peucker algorithm is to derive a simplified polyline to which the perpendicular distance of every point in the original polyline is at most δ. The algorithm initially constructs the line segment (q_1, q_n). It then identifies the point q_i furthest to the line. If this point's perpendicular distance to the line is within δ, it returns (q_1, q_n) and terminates. Otherwise it recursively applies the same process on the two sub-polylines $\{q_1, \cdots, q_i\}$ and $\{q_i, \cdots, q_n\}$.

After simplification, the original polyline is completely lying within the *δ-boundary* of the simplified polyline, and this implies if query trajectory Q is the original polyline, then Q's *ϵ-buffer* is completely within the *$(\sqrt{2}\epsilon + \delta)$-boundary* of the simplified polyline Q', or formally, $\mathcal{B}_\epsilon(Q) \subseteq \mathcal{D}_{\sqrt{2}\epsilon+\delta}(Q')$.

$\mathcal{D}_{\sqrt{2}\epsilon+\delta}(Q')$ can thus serve as an acceptable boundary for grouping work. In specific, for every simplified segment, we start grouping all those consecutive points from its start point aggressively until the next to be grouped area surpasses the boundary. After grouping, if there are still some points between the simplified line segment left ungrouped, we then start the next grouping process until all points end up within their own grouped areas. Algorithm 2 shows this adaptive grouping strategy.

Algorithm 2: *generateGroupedAreas(Q, ϵ, δ)*

```
Input: query trajectory $Q$, threshold $\epsilon$, threshold $\delta$
Output: A list of grouped areas
1  $List_\mathcal{G} \leftarrow \phi$;
2  $Q' \leftarrow DouglasPeucker(Q[1:m], \delta)$;
3  $j \leftarrow 2$;
4  $rect \leftarrow [(+\infty, +\infty), (-\infty, -\infty)]$;
5  for $i \leftarrow 1$ to $m - 1$ do
6      if $i < Q'[j].subscript$ then
7          /* Grouping in the same simplified segment*/
8          $rect_{curr} \leftarrow$ rectangular area of $Q[i]$;
9          $rect_{temp} \leftarrow$ grouped area of $rect_{curr}$ and $rect$;
10         if $rect_{temp}$ within $\mathcal{D}_{\sqrt{2}\epsilon+\delta}(Q')$ then
11             $rect \leftarrow rect_{temp}$;
12         else
13             $List_\mathcal{G}.add(rect)$;
14             /* Start the next grouping process */
15             $rect \leftarrow$ rectangular area of $Q[i]$;
16     else
17         $List_\mathcal{G}.add(rect)$;
18         /* Start grouping the next simplified segment */
19         $j \leftarrow j + 1$;
20         $rect \leftarrow$ rectangular area of $Q[i]$;
21 $List_\mathcal{G}.add$(rectangular area of $Q[m]$); // The last one
22 return $List_\mathcal{G}$;
```

In Algorithm 2, line 10 checks whether the *next to be grouped area $rect_{temp}$* is within the *$(\sqrt{2}\epsilon + \delta)$-boundary* of Q'. It does so by checking whether the perpendicular distances of the four vertices in $rect_{temp}$ to the current simplified line segment $(Q'[j-1], Q'[j])$ are all within $\sqrt{2}\epsilon + \delta$. If yes, then it returns *true*; otherwise it returns *false*.

As an example, for $Q = \{q_1, \cdots, q_9\}$, Figure 5 shows the grouping results, namely $\mathcal{G}(q_1, q_5), \mathcal{G}(q_6, q_8)$ and $\mathcal{G}(q_9, q_9)$, represented by the three red-edged rectangles. Also, the black line $\{q_1, q_6, q_9\}$ in the figure represents the simplified polyline Q' while the black dashed boundary depicts the corresponding $(\sqrt{2}\epsilon + \delta)$*-boundary*.

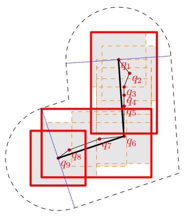


Figure 5: An example of grouped areas

In the best case, we can group all points between a simplified line segment into one grouped area (such as the $\mathcal{G}(q_1, q_5)$ in Figure 5); while in the worst case, we cannot group any two consecutive points at all. Fortunately, the worst case is rare due to the loose boundary derived from line simplification. Overall, the grouping effectiveness is influenced by a variety of factors, including consecutive points distribution, underline trajectory data distribution, and in particular, the value of the distance threshold. Heuristically, setting δ to $\epsilon/2$ achieves good grouping effectiveness as confirmed by the experiments.

According to the result in Figure 5, we now only need to conduct 3 times of range search instead of the originally 9 times to produce the no-false-dismissal candidate set. This, obviously, greatly reduces the search cost as the introduced dead space is much smaller than the repetitive searched overlapping areas. Since we solve the problem caused by overlapping search, we can slightly modify Algorithm 1 to adopt the new *adaptive filter strategy*. Specifically, we call Algorithm 2 between lines 3 and 4 to produce a list of grouped areas $List_{\mathcal{G}}$, replace line 4 with "**foreach** *rect in* $List_{\mathcal{G}}$ **do**" and delete line 5 to get the new algorithm.

3.2 A Better Refinement Strategy

For the refinement phase in Algorithm 1, it simply retrieves every candidate trajectory from the external memory and conducts the quadratic $HCSS$ computation w.r.t. Q to get the most similar trajectory, which, in some extent, bears heavy IO and computation cost. Also, comparing with the original filter strategy, the adaptive filter strategy although successfully reduces the overall range search cost, but it also raises the chance to return more candidate trajectories owing to search the introduced dead space. This may further increase the cost of the refinement phase. Is it possible to reduce this cost?

Observation 3 *Among the large number of returned candidate trajectories in filter phase, only* one *or* a few *trajectories have the chance to be the most similar trajectory.*

This observation suggests us designing a better refinement strategy is not impossible. Such as if we develop an upper bound for $HCSS$, we then can prune most of the unrelated trajectories.

From the adaptive grouping strategy in previous section, after grouping, we know every point in the query trajectory belongs to one and only one grouped area, although between grouped areas some small overlaps may exist. For one grouped area, let k be the number of grouped consecutive points, and let w be the largest weight among the k corresponding weights. Then the product value $k * w$ is called the *group weight* or denoted as $\mathcal{W}$. Each grouped area carries its own *group weight*. For example, the first grouped area $\mathcal{G}(q_1, q_5)$ in Figure 5 grouped five points, if we assume the largest weight among the five corresponding weights is 0.7, then we have $k = 5$, $w = 0.7$, and $\mathcal{G}(q_1, q_5)$ carries the group weight $\mathcal{W} = 5 * 0.7 = 3.5$.

Definition 4. ***(HGSS)*** For any T in the trajectory database, given Q and the corresponding grouped areas $\{\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_K\}$, we define the *Heaviest Grouped Subsequence* between them

$$HGSS(T, Q) = \sum_{i=1}^{K} F(i)$$

where $F(i) = \mathcal{G}_i.\mathcal{W}$ if T has at least one sample points in grouped area $\mathcal{G}_i$, otherwise $F(i) = 0$.

Lemma 1. *$HGSS$ is an upper bound of $HCSS$, that is, $HGSS(T, Q) \geq HCSS(T, Q)$.*

Proof. Let $Q = \{q_1, q_2, \cdots, q_m\}$, then $HCSS(T, Q) = \sum_{i=1}^{m} f(i)$, where $f(i) = w_i$ if during the computation process q_i contributes w_i to the final $HCSS$ value, otherwise $f(i) = 0$. Consider any grouped area $\mathcal{G}_j(q_s, q_e)$, if T has one sample point within this area, then

$$F(j) = \sum_{i=s}^{e} max\{w_s, w_{s+1}, \cdots, w_e\} \geq \sum_{i=s}^{e} w_i \geq \sum_{i=s}^{e} f(i)$$

otherwise $F(j) = 0 = \sum_{i=s}^{e} f(i)$. Thus, we have $\sum_{i=1}^{K} F(i) \geq \sum_{i=1}^{m} f(i)$. □

Since the $HGSS$ can be obtained in the filter phase with negligible cost, we now can design a better strategy for the refinement process.

As shown in Algorithm 3, in the refinement step, we sort all returned candidate trajectories in descending order by $HGSS$ value (line 12), and then visit the first element and compute the exact $HCSS$ value (lines 15-16). If it no less than the $HGSS$ of the next unvisited trajectory (lines 20-22), we return the one with largest $HCSS$ value among all visited trajectories as the most similar trajectory and then terminate, as the $HCSS$ values of the rest trajectories are no larger than the current $HCSS$ value; otherwise we continue this process with the next unvisited trajectory.

Integrated with the adaptive filter strategy and the better refinement strategy, Algorithm 3 is the final optimized algorithm to our user oriented trajectory similarity search.

4. THEORETICAL ANALYSIS

Our upcoming experiments show that our method can efficiently handle the query process. In this section, we do a theoretical analysis to demonstrate the complexity of the proposed method.

Let the database contain N trajectories and every trajectory on average has the length of n sample points. Let the query

Algorithm 3: *OptimizedMST(TR, Q, ϵ, δ)*

```
Input: R-tree root $TR$, query trajectory $Q$, threshold $\epsilon$,
       threshold $\delta$
Output: The unique id of the most similar trajectory
1  $id \leftarrow \infty$;
2  /* Filter phase */
3  $List_{\mathcal{C}} \leftarrow \phi$; // the candidate set
4  $List_{\mathcal{G}} \leftarrow generateGroupedAreas(Q, \epsilon, \delta)$;
5  foreach $rect$ in $List_{\mathcal{G}}$ do
6      $List_{\mathcal{P}} \leftarrow TR$.rangeSearch($rect$);
7      foreach sample point $p$ in $List_{\mathcal{P}}$ do
8          $tid \leftarrow p.getTid()$;
9          if $List_{\mathcal{C}}$.contain(tid) is not ture then
10             $List_{\mathcal{C}}$.add($tid$);
11 /* Refinement phase */
12 Sort $List_{\mathcal{C}}$ in descending order by $HGSS$;
13 $distance \leftarrow -\infty$;
14 for $i \leftarrow 1$ to $List_{\mathcal{C}}.length$ do
15     $T \leftarrow$ retrieve the $List_{\mathcal{C}}[i].tid$th trajectory from external
       memory;
16     $hcss \leftarrow HCSS(T, Q)$;
17     if $distance < hcss$ then
18         $distance \leftarrow hcss$;
19         $id \leftarrow tid$;
20     if $hcss \geq List_{\mathcal{C}}[i+1].HGSS$ then
21         /* Prune the rest trajectories */
22         break;
23 return $id$;
```

trajectory have the length of m. Let the average cost retrieving one trajectory from the database be C. As the similarity value $HCSS$ is derived in a dynamic programming manner, its computation cost is $O(mn)$. For the naive method, as it has to retrieve every trajectory from the database and compute the exact $HCSS$ value, its complexity consists IO cost $O(NC)$ as well as computation cost $O(Nmn)$, namely $O(N(C + mn))$.

Let C_R be the average range search cost on R-tree; let ξ_1 be the candidate trajectory rate. The complexity of Algorithm 1 consists two parts: filter cost $O(mC_R)$ and refinement cost $O(\xi_1 N(C + mn)$, therefore it is $O(mC_R + \xi_1 N(C + mn))$.

Let ξ_2 be the actually processed trajectory rate after pruning; let m' be the number of grouped areas. Obviously, $\xi_2 < \xi_1$ and $m' < m$. As in the adaptive filter strategy we bound the introduced dead space, we expect the average range search in Algorithm 3 is almost the same with C_R, or at most constant times of C_R, therefore the filter cost of Algorithm 3 is $O(m'C_R)$, and the complexity of it is $O(m'C_R + \xi_2 N(C + mn))$.

To sum up, the complexities of the three algorithms are shown in Table 1.

Generally, the range searching cost is far less than the cost of retrieval all trajectories from database, i.e., $mC_R \ll NC$, and the rates ξ_1 and ξ_2 are also expected to be small, such as less than 10%, then Algorithm 1 and 3 are far more efficient than the naive method, as confirmed by our experiments.

Table 1: Algorithm complexity

Algorithm	Complexity
$Naive$	$O(N(C + mn))$
MST	$O(mC_R + \xi_1 N(C + mn))$
$OptimizedMST$	$O(m'C_R + \xi_2 N(C + mn))$

5. EXPERIMENTS

In this section, we refer Algorithm 1 as MST, Algorithm 3 as $OMST$, and the naive method as $Naive$. We then design experiments to answer the following questions:

1. Compared with the $Naive$ method, what are the performance of our methods: MST, $OMST$?
2. In terms of saved IO cost, what is the performance of the adaptive filter strategy?
3. In terms of saved cost (both IO and computation), what is the performance of the better refinement strategy?

We measure the IO cost in question 2 with the number of disk blocks that the algorithm visits during the range search; meanwhile, we measure the saved cost in question 3 with the number of trajectories retrieved from disk and processed during the refinement phase.

Since the matching threshold ϵ is application dependent [13], we run several probing programs on each dataset and choose the one close to human observations. Also, we set the distance threshold δ as $\epsilon/2$.

5.1 Settings

Dataset: We use Beijing dataset for the experiment. The dataset is a three-day taxi trajectory dataset whose distribution is shown in Figure 6. After cleaning, the dataset contains 6176841 sample points and consists of 30284 trajectories with lengths varying from 20 to 1400.

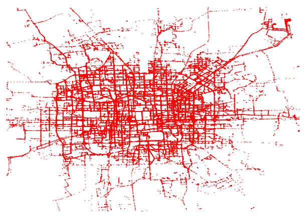


Figure 6: The Beijing Dataset

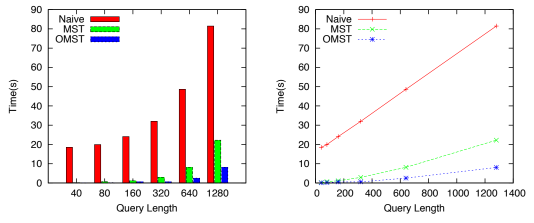
To mimic the scenario that the dataset is too large to resident in main memory, we put the dataset in external memory and use an R-tree to index it. In the R-tree, we set the page size 4096 bytes, the capacities and the fill factors (both node and leaf) 100 and 70%, respectively.

The query trajectories are divided into six groups whose lengths are 40, 80, 160, 320, 640, 1280, respectively. Each group consists of 50 trajectories and the cost is obtained from the average of the corresponding items. For each query trajectory, we use a random function to weight its sample points.

Environments: All these algorithms are implemented in Java and examined on a Windows XP platform with Intel Core i7 CPU (2.93GHz) and 3.5GB memory.

5.2 Performance

Figure 7(a) shows the query time of the three algorithms in histogram. From it we can see the $Naive$ method takes the longest time for each query. As a matter of fact, even for queries as short as 40, it still takes about 20 seconds to get the answer. While for our proposed method MST, the answering time is significantly shorter. Even for the longest queries, it only takes about 20 seconds to get the answer. The optimized $OMST$ algorithm is the most efficient algorithm. Its answering time is significantly shorter than the corresponding time of MST and drastically shorter than the corresponding time of $Naive$.



(a) (b)

Figure 7: The query time

Figure 7(b) shows the query time in the real length scale. As the computation cost of the similarity value is quadratic to the query length, if we only count on the computation cost, the query time should be quadratic to the length. But the real time is nearly linear. This implies that the IO cost plays an important part in the query process.

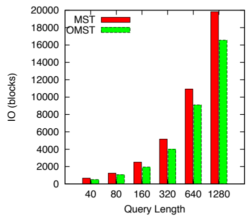


Figure 8: Range query efficiency

Figure 8 illustrates the number of disk blocks that the two algorithm visits during the filter phase with the range queries. Since algorithm $OMST$ adopts the adaptive filter strategy, from the figure, we can see its IO cost is significantly smaller than the corresponding cost in MST, which implies the adaptive filter strategy is quite effective.

In Figure 9, the MST denotes the number of trajectories in the candidate set in Algorithm 1 and also the number of processed trajectories in the refinement phase; the $OMST$ denotes the number of trajectories in the candidate set with the adaptive filter strategy in Algorithm 3 and the $OMST$-*prune* denotes the number of actually processed trajectories in the refinement phase after pruning. We can see the number of trajectories of $OMST$ is significantly higher than that

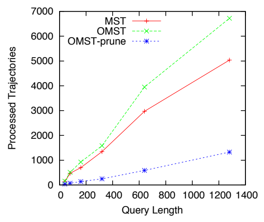
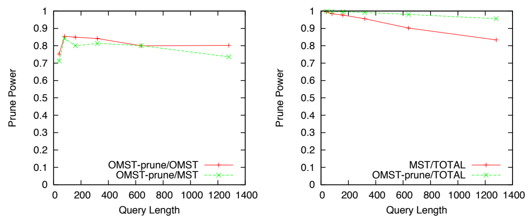


Figure 9: Processed trajectories

of MST due to the introduced dead space in the adaptive filter strategy, while after pruning, the number of actually processed trajectories is much smaller than that of MST and $OMST$, which implies the pruning ability of the upper bound is quite good.



(a) (b)

Figure 10: The prune power

Figure 10(a) shows the relative prune power of the pruning techniques over MST and $OMST$. The relative prune power is defined to be the fraction of candidate trajectories being pruned in the refinement phase. We can see both the relative pruning powers in MST and $OMST$ are around 80%, which from another perspective demonstrates the effectiveness of our pruning techniques. Figure 10(b) shows the absolute prune power of Algorithm 1 and Algorithm 3, where the absolute prune power is the fraction of trajectories being pruned in the refinement phase w.r.t. all trajectories in the database. For Algorithm 1, the prune power is above 83%; while for Algorithm 3, the prune power is above 95%. Having the ability to prune most trajectories in the database is the main reason that Algorithm 1 and Algorithm 3 are far more efficient than the Naive method.

6. RELATED WORK

Trajectory similarity search has been studied for decades. From the earliest similarity measures like Euclidean-based distance [1, 15, 11], Dynamic Time Warping (DTW) [16, 9, 4] to the more recently similarity measures like Edit Distance on Real sequence (EDR) [3], Longest Common Subsequence (LCSS) [13], a consider amount of methods have been proposed. Among these methods, Euclidean-based distance and DTW are sensitive to noise, while the EDR and LCSS are more robust and accurate. However, none of these work considers user oriented similarity search, which is exactly what we study in this paper.

In [13] Vlachos et al. suggest using LCSS as the similarity measure, which matches two sequences by allowing them to stretch, without rearranging the sequence of elements but allowing some elements to be unmatched. As a result, the LCSS measure can efficiently handle outliers (or noise) that often exist in trajectories due to sensor failures, error in detection techniques and disturbance signals. A cluster-based indexing is proposed to improve the retrieval efficiency using LCSS. The performance of this indexing method depends on the clustering results. However, due to LCSS not following triangle inequality, it is hard to find good clusters and representing points in the data set [8]. Besides, [13] considers shape similarity allowing spatial shifting between trajectories, which is quite different form our settings: the spatial shifting is strictly prohibited.

Chen et al. in [3] propose EDR as a similarity measure which considers spatial shifting as well as assigning penalties according to the sizes of gaps in between similar shapes. Three pruning techniques are also developed to speed up the query process. But under the condition of user specified query where the relative importance of each sample point is considered, the pruning techniques are no longer hold.

In the query trajectory, if we set all the significant points as weight 1, and other points as weight 0, then this work bears some resemblance to [5] in which Chen et al. explore searching trajectories by specifying a series locations. However,as the number of specified locations may be a few, in our work, we study a more general problem with arbitrary number of sample points (theoretically) where each sample point may pertain to arbitrary weight.

There are also some interesting work [20, 19, 18, 17, 10] using data mining techniques to find the semantic aspects of trajectories. In these work, each trajectory is firstly transformed into a semantic trajectory, then based on semantic representing the system does friends recommendation. Different from these work, our work towards providing personalized recommendation in which the preference (or significance) is designated by users.

7. CONCLUSION

In this paper, we study a new problem of user-oriented trajectory similarity search, in which each user can specify their own important parts in the query trajectory to get the personalized searching results. We identify the efficiency issue as the key challenge and develop a two-phase algorithm taking advantage of the spatial localities to conquer this challenge. As we observe that there are some redundant IOs in the filter phase and only few trajectories have the chance to be returned in the refinement phase, we develop two optimized strategies to speed up the query process. The theoretical analysis and the experiment results demonstrate the effectiveness of the two optimized strategies and justify the advantage of the proposed methods over the naive method for at least an order of magnitude. Furthermore, the proposed methods are easy to implement and incorporate into trajectory databases.

8. ACKNOWLEDGMENTS

This paper is partially supported by NSFC project under grant number 60970030.

9. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. *FODO*, pages 69–84, 1993.

[2] L. Chen and R. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[3] L. Chen, M. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, pages 491–502. ACM, 2005.

[4] S. Chen and R. Kashyap. A spatio-temporal semantic model for multimedia database systems and multimedia information systems. *TKDE*, 13(4):607–622, 2001.

[5] Z. Chen, H. Shen, X. Zhou, Y. Zheng, and X. Xie. Searching trajectories by locations: an efficiency study. In *SIGMOD*, pages 255–266. ACM, 2010.

[6] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM, 1984.

[8] D. Jacobs, D. Weinshall, and Y. Gdalyahu. Classification with nonmetric distances: Image retrieval and class representation. *PAMI*, 22(6):583–600, 2000.

[9] E. Keogh. Exact indexing of dynamic time warping. In *VLDB*, pages 406–417, 2002.

[10] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W. Ma. Mining user similarity based on location history. In *GIS*, pages 34:1–34:10. ACM, 2008.

[11] B. Lin and J. Su. Shapes based trajectory queries for moving objects. In *GIS*, pages 21–30. ACM, 2005.

[12] A. Palma, V. Bogorny, B. Kuijpers, and L. Alvares. A clustering-based approach for discovering interesting places in trajectories. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 863–868. ACM, 2008.

[13] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684. IEEE, 2002.

[14] X. Xiao, Y. Zheng, Q. Luo, and X. Xie. Finding similar users using category-based location history. In *GIS*, pages 442–445. ACM, 2010.

[15] Y. Yanagisawa, J. Akahani, and T. Satoh. Shape-based similarity query for trajectory of mobile objects. In *MDM*, pages 63–77. Springer, 2003.

[16] B. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208. IEEE, 1998.

[17] J. Ying, E. Lu, W. Lee, T. Weng, and V. Tseng. Mining user similarity from semantic trajectories. In *LBSN*, pages 19–26. ACM, 2010.

[18] Y. Zheng and X. Xie. Learning travel recommendations from user-generated gps traces. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(1):2, 2011.

[19] Y. Zheng, L. Zhang, Z. Ma, X. Xie, and W. Ma. Recommending friends and locations based on individual location history. *TWEB*, 5(1):5, 2011.

[20] Y. Zheng, L. Zhang, X. Xie, and W. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, pages 791–800. ACM, 2009.