Searching Trajectories by Locations – An Efficiency Study

Zaiben Chen[†], Heng Tao Shen[†], Xiaofang Zhou[†], Yu Zheng[‡], Xing Xie[‡]
[†] School of Information Technology & Electrical Engineering
The University of Queensland, QLD 4072 Australia
[‡]Microsoft Research Asia, Beijing 100080 China
{zaiben, shenht, zxf}@itee.uq.edu.au, {yuzheng, xingx}@microsoft.com

ABSTRACT

Trajectory search has long been an attractive and challenging topic which blooms various interesting applications in spatial-temporal databases. In this work, we study a new problem of searching trajectories by locations, in which context the query is only a small set of locations with or without an order specified, while the target is to find the k *Best-Connected Trajectories* (k-BCT) from a database such that the k-BCT best connect the designated locations geographically. Different from the conventional trajectory search that looks for similar trajectories w.r.t. shape or other criteria by using a sample query trajectory, we focus on the goodness of connection provided by a trajectory to the specified query locations. This new query can benefit users in many novel applications such as trip planning.

In our work, we firstly define a new similarity function for measuring how well a trajectory connects the query locations, with both spatial distance and order constraint being considered. Upon the observation that the number of query locations is normally small (e.g. 10 or less) since it is impractical for a user to input too many locations, we analyze the feasibility of using a general-purpose spatial index to achieve efficient k-BCT search, based on a simple *Incremental k-NN based Algorithm* (IKNN). The IKNN effectively prunes and refines trajectories by using the devised lower bound and upper bound of similarity. Our contributions mainly lie in adapting the *best-first* and *depth-first* k-NN algorithms to the basic IKNN properly, and more importantly ensuring the efficiency in both search effort and memory usage. An in-depth study on the adaption and its efficiency is provided. Further optimization is also presented to accelerate the IKNN algorithm. Finally, we verify the efficiency of the algorithm by extensive experiments.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms, Performance

Keywords

Trajectory Search, Locations, Efficiency, Optimization

1. INTRODUCTION

The proliferation of mobile devices enables people to log their geographical positions and to trace historical movements, which has spawned various novel applications. An emerging one is trajectory sharing and searching. Different from the conventional similarity search [22, 21, 8, 20] over trajectory databases that uses a sample query trajectory for full matching according to shape or other criteria, new trajectory search applications demand to find trajectories that connect a few selected locations, i.e., are close to all the locations. As exemplified in Figure 1, given a set of coordinates of locations (represented by dots) acquired by clicking on map or image geo-coding, we retrieve from a database the raw trajectory (indicated by the marked track) that best connects these locations. This query can benefit travelers when they are planning a trip to multiple places of interest in an unfamiliar city by providing similar routes traveled by other people for reference.

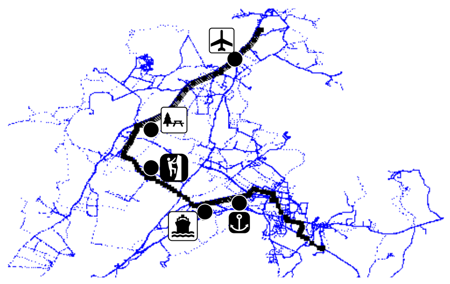

Figure 1: Trajectory search by locations – an example

Moreover, by utilizing this new query, a tourist/travel agency can further survey popular routes through some attractions; a zoologist is capable of finding the nearest trajectories of animals to some stationary points (e.g. source of food) [11]; a transport department is able to investigate the driving habits of local residence within a suburb and onwards analyze the traffic causes of some critical road intersections, etc. This work is based on the Microsoft GeoLife project [1], and some example websites for sharing and searching trajectories include Bikely, GPS-Waypoints and

Share My Routes[1], that all provide a platform for people to share trajectories of driving, hiking, boating etc., and some very simple search functions have already been developed.

In this work, we propose a new type of trajectory query called the *k Best-Connected Trajectory* (k-BCT) query for searching trajectories by multiple geographical locations. Generally, the k-BCT query is a set of locations indicated by coordinates like $(latitude, longitude)$, which could be a famous attraction, a nameless beach, or any arbitrary place approximated by the center location. The user may also specify a preferred visiting order for the intended places, in which case the order of a trajectory needs to be taken into account as well. For example, a query with three locations A, B and C appears as

$$\{A_{(37.2601,122.0941)}, B_{(37.2721,122.0648)}, C_{(37.3344,122.1538)}\}$$

where $(37.2601, 122.0941)$ denotes the $(latitude, longitude)$ of A. If an order constraint further applies, the found trajectories connecting A, B and C should also preserve the visiting order ($A \rightarrow B \rightarrow C$). Obviously, the conventional similarity search [22, 21, 8, 20, 9] can not address this problem since the similarity measure is different in our applications, which should reveal the connectivity between a trajectory and the query locations rather than the similarity in shape. Furthermore, the query is no longer a full trajectory or any sub-sequence, but a few arbitrary locations, which makes the query more flexible. In essence, the k-BCT query can be treated as an extension of the traditional single point-based query [11, 18] which looks for the nearest trajectories to only one location, while the k-BCT query searches the 'nearest' trajectories to multiple locations.

To answer the k-BCT query, we can separately issue at each query location an independent single point-based query to find out the nearest trajectories w.r.t. each location, and then the trajectories within the intersection of query results, if exist, are supposed to be close to all the query locations and thus with a good connectivity. Based on this basic idea, we propose an *Incremental k-NN based Algorithm* (IKNN), which retrieves the nearest trajectory points w.r.t. each query location incrementally and examines the k-BCT from the trajectory points discovered so far. In this method, the pruning and refinement of the search are conducted by using the lower bound and upper bound of trajectory similarity that are derived from the found trajectory points, and the retrieval of nearest trajectory points is based on the traditional *best-first* and *depth-first* k-NN algorithms [19, 14] over an R-tree [13] index. Note that the utilization of the commonly used R-tree is only for browsing distance between points. Other spatial indexes can also be easily adapted.

However, the problem of the basic IKNN algorithm that we concern is about the search efficiency, which originates from the efficiency of retrieving the nearest trajectory points incrementally. For the *best-first* k-NN algorithm [14], it provides an I/O optimal solution but causes a lack of guarantee in memory usage, which means it may cause an overlarge memory consumption that could potentially bring down the system or severely affect the number of concurrent queries. For the *depth-first* k-NN algorithm, it's consumption of memory is ascertained, but it performs very poorly in the IKNN algorithm as it involves too much repetition of search regions in our case. Therefore, besides proposing the new query as well as a new similarity function, we theoretically analyze the cost of the IKNN adopting these two k-NN algorithms respectively, and a major part of our contribution lies in the adaption of the k-NN algorithms to the IKNN. Through the adaption mechanism, the IKNN adopting the *depth-first* strategy achieves a good search performance comparable to the *best-first* strategy in terms of R-tree node access and query time, while it also guarantees a fixed low memory consumption. Moreover, we also present an optimization solution to conduct the IKNN search over multiple query locations smartly, and it achieves notable improvement.

The remainder of the paper is organized as follows. In Section 2 we discuss the related work. In Section 3, the formal definition of the k-BCT query and its similarity function are given. The IKNN algorithm is presented in Section 4 with detailed analysis. Finally we show our experiment results in Section 5 and draw a conclusion in Section 6.

2. RELATED WORK

Our work shares a common methodology with similarity search on trajectory/time series data. Basically the first step is to define a similarity/distance function by some kind of aggregation of distances between trajectory points, and then efficient query processing algorithms are needed to address the problem of searching over a large set of candidate trajectories. A considerable amount of related work has been proposed before. Several typical similarity functions for different applications include Euclidean Distance [2], Dynamic Time Warping (DTW) [22], Longest Common Subsequence (LCSS) [21], Edit Distance with Real Penalty (ERP) [7], Edit Distance on Real Sequences (EDR) [8], and enhanced techniques for evaluating the similarity of time series are also studied in [16, 20].

The pioneering work by Agrawal et al. in [2] utilizes Discrete Fourier Transform (DFT) to transform trajectories to multi-dimensional points, and then the comparison between trajectories can be done by comparing the Euclidean distance between the points in the feature space, which should match or under-estimate the real distance. Faloutsos et al. extend this work in [10] to allow subsequence matching. Other transform functions such as Discrete Wavelet Transform (DWT) [6] are also available. Cai et al. [5] utilize the Chebyshev polynomials for approximating and indexing trajectories. However, those methods require the time series to have the same length and the transform functions are not applicable for our query.

Frentzos et al. define in [12] a distance measure as the area of the region between two trajectories. Nevertheless, their method does not work if the two trajectories are of different lengths. In contrast, DTW [22] adopts time-shifting in the comparison of trajectories and the basic idea is to allow 'repeating' some points as many times as needed to achieve the best alignment. Yet, all points including noises have to be matched, and therefore it may accumulate an over large DTW distance merely due to a single noisy point. Compared with DTW, LCSS [21] allows to 'skip' some points other than re-arrange the order of these points. As a consequence, far-away points would be ignored and thus it is robust to noises. However, it needs a matching threshold to determine whether to take a point into account. Chen et al. propose in [8] the EDR distance, which is similar to LCSS in using a threshold parameter to determine if two points are

[1]Bikely (http://www.bikely.com/)
GPS-Waypoints (http://www.gps-waypoints.net/)
Share My Routes (http://www.sharemyroutes.com/)

matched in comparison, while considering penalties to gaps. In [7], Chen et al. also propose the ERP distance aiming to combine the merits of DTW and EDR, by using a constant reference point for computing distance.

A similarity function is normally designed by considering the actual application, and none of the similarity functions above satisfies the requirements of our applications, in which the query is broken into a small set of locations, and we concern more on whether a trajectory provides a good connection to query locations rather than whether the trajectory is similar to the query in shape. Therefore, due to the new application requirements, we need to define a new similarity function. For query processing, this is the first work on trajectory search by multiple query locations. As the number of query locations is typically small, we are able to use a spatial method for the search. Here, our IKNN algorithm utilizes the k-NN algorithms in [19] and [14] over an R-tree [13] index of all the trajectory points. Notice that we do not consider the STR-tree or TB-tree [18] for trajectory indexing and searching since they focus more on trajectory preservation and leave other spatial properties like spatial proximity aside, while in the IKNN algorithm, the R-tree index is only for fast retrieval of nearest points.

3. THE K-BCT QUERY

A trajectory database contains a large collection of raw trajectories recorded by GPS devices in the form of a series of locations/points $\{p_1, p_2, \cdots, p_l\}$, where p_i is a trajectory point represented by $(latitude, longitude)$, and l is the number of points in the trajectory. The k *Best-Connected Trajectory* (k-BCT) query Q, according to our applications, is represented by a set of locations:

$$Q = \{q_1, q_2, \cdots, q_m\}$$

m is the number of query locations, and the user may further assign a visiting order to the locations, in which case Q is treated as a sequence of locations from q_1 to q_m. Notice that we use the term 'location' and 'point' interchangeably.

In order to search for the best-connected trajectories from a database, firstly we need a similarity function to score how well a trajectory connects the query locations, which should consider the distance from the trajectory to each query location. We simply define the distance $Dist_q$ between a query location q_i and a trajectory $R = \{p_1, p_2, \cdots, p_l\}$ as:

$$Dist_q(q_i, R) = \min_{p_j \in R}\{Dist_e(q_i, p_j)\}$$

$Dist_e(q_i, p_j)$ is the Euclidean distance between q_i and p_j, so actually $Dist_q(q_i, R)$ is the shortest distance from q_i to any point on R. Here we call the $< q_i, p_j >$ a *matched pair* where p_j is the nearest point on R to q_i. Then for a query Q that consists of $\{q_1, q_2, \cdots, q_m\}$ without an order constraint, we define the similarity $Sim(Q, R)$ between Q and R based on the distance of each matched pair as follows:

$$Sim(Q, R) = \sum_{i=1}^{m} e^{-Dist_q(q_i, R)} \qquad (1)$$

The intuition of using the exponential function $e^{-Dist_q(q_i, R)}$ to measure the *contribution* of each matched pair to $Sim(Q, R)$ is that we would like to assign a larger contribution to a closer matched pair of points while giving a much lower value to those far away pairs, which results in an exponential decreasing of contribution as $Dist_q(q_i, R)$ linearly increases. Therefore, only a trajectory that is really close to all the query locations is considered to be 'similar', which agrees with the perception of human beings. A matching example is illustrated in Figure 2(a), where the query locations q_1, q_2 and q_3 are matched to the closest points p_6, p_4 and p_7 on R respectively. Therefore, $Sim(Q, R) = e^{-Dist_e(q_1, p_6)} + e^{-Dist_e(q_2, p_4)} + e^{-Dist_e(q_3, p_7)} = e^{-1.5} + e^{-0.1} + e^{-0.1}$. The dashed ellipses here indicate the matched pairs.

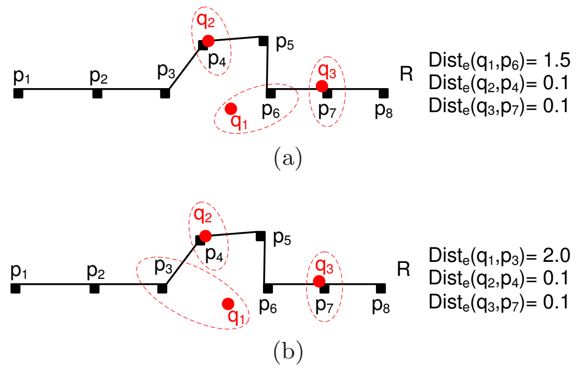

Figure 2: Matching a trajectory

When a visiting order is further specified for the query locations, the matched points on R have to preserve the order. Thus, an adjustment of matching may be necessary, and the matched point p_j for location q_i may not be the nearest point to q_i any longer. Considering the example in Figure 2(a) again, suppose the user wishes to make visits in the order of $q_1 \rightarrow q_2 \rightarrow q_3$. The actual order of the matched points on R is $p_4 \rightarrow p_6 \rightarrow p_7$ (assume R goes from left to right), rather than $p_6 \rightarrow p_4 \rightarrow p_7$, therefore this matching does not conform with the user-specified order, and hence we need to adjust the matching of trajectory points to satisfy the order constraint. As shown in Figure 2(b), q_1 is rematched with p_3 and then the new visiting order is $p_3 \rightarrow p_4 \rightarrow p_7$ which correctly preserves the user-specified order. Here, our target is to maximize the sum of the contribution of each matched pair, while still keeping the order of visits, i.e., the sum of the contribution of $< q_1, p_3 >$, $< q_2, p_4 >$ and $< q_3, p_7 >$ pairs is maximum among all the possible combinations that preserve the order.

However, the calculation of similarity is not straightforward when considering a visiting order. Given a sequence of query locations $Q = \{q_1, q_2, \cdots, q_m\}$, and a trajectory $R = \{p_1, p_2, \cdots, p_l\}$, we define the similarity $Sim_o(Q, R)$ for ordered query locations in a recursive way as follows:

$$Sim_o(Q, R) = \max \begin{cases} e^{-Dist_e(Head(Q), Head(R))} \\ +Sim_o(Rest(Q), R) \\ \\ Sim_o(Q, Rest(R)) \end{cases} \qquad (2)$$

where $Head(*)$ is the first point of $*$, e.g., $Head(Q) = q_1$, and $Rest(*)$ indicates the rest part of $*$ after removing the first point, e.g., $Rest(Q) = \{q_2, q_3, \cdots, q_m\}$. The idea of Equation 2 is to define the maximal solution to $Sim_o(Q, R)$ recursively in terms of the maximal solutions to subproblems: $Sim_o(Rest(Q), R)$ and $Sim_o(Q, Rest(R))$. Therefore, once $Head(Q)$ and $Head(R)$ match, we sum up $e^{-Dist_e(Head(Q), Head(R))}$ to the similarity and shift to the matching of the rest of Q by calling $Sim_o(Rest(Q), R)$. In

Table 1: A list of notations

Notation	Explanation
N	The number of all the trajectory points
m	The number of query locations
$Dist_e(q_i, p_j)$	The Euclidean distance between q_i, p_j
$Dist_q(q_i, R)$	The shortest distance from q_i to R
C	The candidate set
c	The fanout of the R-tree
ϵ	The radius of a 'shape'
$\overline{nb}(\epsilon, \text{'shape'})$	The average number of neighbors within a region of 'shape' and radius ϵ
D_2, D_0	The Fractal Dimensions, $D_2 = D_0 = 2$ for a uniform distribution
r	The radius of a λ-NN search region
σ_j	The average side of MBR at level j
$Dist_{last}$	The distance of the previous λ^{th} NN
MBR$_1$	The first visited MBR that contains not less than λ points in a *depth-first* search
ρ	The density of trajectory points
$\xi(q_i)$	The contribution of q_i to upper bound
μ, ν	The weights for optimization

this case, $Head(R)$ is still retained for the next round of comparison as a trajectory point may be matched with more than one query locations. Besides that, we can also skip trajectory points by calling $Sim_o(Q, Rest(R))$. Essentially this is a dynamic programming way to figure out the similarity, while keeping the matched trajectory points in the same order as the query locations. Obviously, Equation 2 combines the merits of DTW [22] which can repeat some trajectory points, and LCSS [21] that can skip un-matched trajectory points including outliers in the matching process. Formally, we define the k-BCT query as below:

Definition 1. (**k-BCT query**) *Given a set of trajectories $T = \{R_1, R_2, \cdots, R_n\}$ $(n \geq k)$, a set of locations Q, and a corresponding similarity function, the k-BCT query is to find the k trajectories $T^{'}$ from T, such that:*

$$Similarity(Q, R_i)_{R_i \in T'} \geq Similarity(Q, R_j)_{R_j \in T - T'}$$

where $Similarity(Q, R_i) = Sim(Q, R_i)$ if no order is specified for Q, otherwise $Similarity(Q, R_i) = Sim_o(Q, R_i)$.

4. QUERY PROCESSING

To answer the k-BCT query, we take advantage of the observation that the number of query locations is usually small. Thus we are able to adopt a spatial index for the search of close-by trajectories for each query location separately, and then merge the results for the exact k-BCT. The total cost of this method is expected to be the cost of searching over the index multiplying the number of query locations, which is a small constant. Here, since knowing the closest points on a trajectory to the query locations is sufficient to estimate the lower bound and upper bound of the proposed trajectory similarity (Equation 1 & 2) for pruning, we adopt the commonly used R-tree [13] index for the search of closest trajectory points. With points of all the database trajectories indexed by one single R-tree, once we find the nearest point w.r.t. a query location, the trajectory that contains this point must be the closest trajectory to the query location. Besides, to support the whole trajectory retrieval, points from the same trajectory are further connected by a double linked list. In the following, we firstly introduce a simple k-NN based method that utilizes the R-tree for the search of k-BCT, followed by an in-depth study on effective adaption and further optimization. Table 1 shows a partial list of symbols used in this paper.

4.1 The IKNN Algorithm

By indexing trajectory points in an R-tree, we are able to efficiently find the closest trajectory point with respect to a query location by using the k-Nearest Neighbor (k-NN) search [19, 14]. Assume there is a query Q comprising of m locations $\{q_1, q_2, \cdots, q_m\}$ without an order constraint, we firstly retrieve the λ-NN of each query location ($\lambda > 0$):

$$\begin{aligned}
&\lambda\text{-NN}(q_1) = \{p_1^1, p_1^2, \cdots, p_1^\lambda\} \\
&\lambda\text{-NN}(q_2) = \{p_2^1, p_2^2, \cdots, p_2^\lambda\} \\
&\cdots \\
&\lambda\text{-NN}(q_m) = \{p_m^1, p_m^2, \cdots, p_m^\lambda\}
\end{aligned}$$

The set of scanned trajectories that contain at least one point in λ-NN(q_i) then form a candidate set C_i for the k-BCT results. Note that the cardinality $|C_i| \leq \lambda$, as there may be several λ-NN points belong to the same trajectory. By merging the candidate sets generated by all the λ-NN(q_i), we get totally f different trajectories as candidates:

$$C = C_1 \cup C_2 \cup \cdots \cup C_m = \{R_1, R_2, \cdots, R_f\}$$

For each trajectory R_x ($x \in [1, f]$) within C, it must contain at least one point whose distance to the corresponding query location is determined. For example, if $R_x \in C_i$ ($C_i \subseteq C$), then the λ-NN of q_i must include at least one point on R_x, and the shortest distance from R_x to q_i is known. Therefore, at least one matched pair of points between R_x and some q_i is discovered, and then a lower bound LB of similarity for each candidate R_x ($x \in [1, f]$) can thereafter be computed by using the found matched pairs:

$$LB(R_x) = \sum_{i \in [1,m] \wedge R_x \in C_i} \left(\max_{j \in [1,\lambda] \wedge p_i^j \in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} \right) \tag{3}$$

Here $\{q_i | i \in [1, m] \wedge R_x \in C_i\}$ denotes the set of query locations that have already been matched with some point on R_x, and the p_i^j which achieves the maximum $e^{-Dist_e(q_i, p_i^j)}$ with respect to q_i is the point on R_x that is closest to q_i, i.e., $\max_{j \in [1,\lambda] \wedge p_i^j \in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} = e^{-Dist_q(q_i, R_x)}$. So $LB(R_x) = \sum_{i \in [1,m] \wedge R_x \in C_i} (e^{-Dist_q(q_i, R_x)})$ and obviously it is no larger than $\sum_{i=1}^{m} e^{-Dist_q(q_i, R_x)}$, because it only takes those matched pairs found so far into account. Thus $LB(R_x)$ must lowerbound the exact similarity $Sim(Q, R_x)$ defined in Equation 1. On the other hand, if $R_x \notin C_i$, then none of it's trajectory points has been scanned by λ-NN(q_i) yet.

For those trajectories that are not contained in C, they have not been scanned by any of the λ-NN search, and any point on them must have a distance to q_i no less than the distance of the λ^{th} NN of q_i (i.e. $Dist_e(q_i, p_i^\lambda)$). Therefore, we can determine an upper bound UB_n of similarity for all the *non-scanned trajectories* as follows:

$$UB_n = \sum_{i=1}^{m} e^{-Dist_e(q_i, p_i^\lambda)} \tag{4}$$

By using these two bounds, we can set up a pruning mechanism for the k-BCT query to avoid scanning the whole trajectory database and thus restrict the search space.

Theorem 1. *For a k-BCT query without an order constraint, if we can get a subset of k trajectories C' from the candidate set C after searching the λ-NN of each query location, and we have $\min_{R_x \in C'}\{LB(R_x)\} \geq UB_n$, then the k best-connected trajectories must be included in C.*

Proof. For any $R_x \in C'$, $Sim(Q, R_x) \geq LB(R_x)$, while for any $R_y \notin C$ (i.e. $R_y \in \overline{C}$), $UB_n \geq Sim(Q, R_y)$. When Theorem 1 is satisfied, which is $\min_{R_x \in C'}\{LB(R_x)\} \geq UB_n$, we can conclude that $\forall R_x \forall R_y (R_x \in C' \wedge R_y \in \overline{C}) \rightarrow (Sim(Q, R_x) \geq Sim(Q, R_y))$. Therefore, no k-BCT result can be from $\overline{C}$, and they must be from C. $\square$

Notice that C' is not necessarily to be the k-BCT, and we can only guarantee the k-BCT results are included in C. Apparently, Theorem 1 provides a way to decide when the k-BCT results are found during the λ-NN search.

Here, one critical question is that what λ we should choose in the search to guarantee that the k-BCT results are contained in the candidate set. If λ is set to be a very large value, it is probably that the k-BCT results will all be retrieved, but the search space will be huge as well. Nevertheless, a smaller λ may not be enough to make sure the k-BCT results are included in C and thus leads to false dismissal. Instead of choosing a fixed λ, we devise an *Incremental k-NN based Algorithm* (IKNN) for efficient retrieval of the candidate trajectories in a filter-and-refine fasion. The basic idea is to search for the λ-NN of each query location first (initially λ can be any positive integer, e.g., $\lambda = k$). If the generated candidate set C does not satisfy Theorem 1, we increase the search region by some Δ and search for the $(\lambda+\Delta)$-NN. This process keeps going until the k-BCT is found. Algorithm 1 shows the details.

In each round of the IKNN (the *while* loop), we first of all retrieve the λ-NN of q_i by KNN(q_i, λ) at line 7, and then construct the candidate set C through line 8 and 9. If we have got enough candidates, then the lower bounds $LB[]$ for all candidates and the upper bound UB_n defined in Formula 3 and 4 respectively are computed at line 11 and 12. The k maximal lower bounds $k\text{-}LB[] \subset LB[]$ are updated at line 13. If Theorem 1 is satisfied at line 14, which means the k-BCT has already been included in C and all the non-scanned trajectories beyond C can be safely filtered out, then the refinement procedure is triggered to examine the exact similarity of candidate trajectories at line 15, after which the k best-connected trajectories are returned. Otherwise we increase λ by Δ and go to the next round.

The refine() function in Algorithm 1 performs a refinement step and it further prunes un-qualified candidates from C to get the final k-BCT results. To achieve this, we further define an upper bound UB of similarity for candidate trajectories in C only, following the same rationale of Equation 3 and 4.

$$UB(R_x) = \sum_{i \in [1,m] \wedge R_x \in C_i} \left(\max_{j \in [1,\lambda] \wedge p_i^j \in R_x} \{e^{-Dist_e(q_i, p_i^j)}\} \right) + \sum_{i \in [1,m] \wedge R_x \notin C_i} (e^{-Dist_e(q_i, p_i^{\lambda})}) \quad (5)$$

In Equation 5, $R_x \in C = \{C_1 \cup C_2 \cdots \cup C_m\}$. For a query location within $\{q_i | i \in [1,m] \wedge R_x \in C_i\}$, the closest point on R_x to q_i is found by the λ-NN(q_i) search, and thus we accumulate to $UB(R_x)$ the contribution of the matched pair $< q_i, closestPoint >$, which is the same as Equation 3. Otherwise, for a q_i that λ-NN(q_i) has not covered any point on R_x (i.e. $R_x \notin C_i$), we consider the current λ^{th} NN of q_i (i.e. p_i^{λ}) that must be closer than the matched point, and accumulate the contribution of the $< q_i, p_i^{\lambda} >$ pair to $UB(R_x)$, which is similar to Equation 4. Straightforwardly, we have:

$$\begin{aligned}
&Sim(Q, R_x) - UB(R_x) = \\
&\textstyle\sum_{i=1}^{m} e^{-Dist_q(q_i, R_x)} - \sum_{i \in [1,m] \wedge R_x \in C_i} (e^{-Dist_q(q_i, R_x)}) \\
&- \textstyle\sum_{i \in [1,m] \wedge R_x \notin C_i} (e^{-Dist_e(q_i, p_i^{\lambda})}) \\
&= \textstyle\sum_{i \in [1,m] \wedge R_x \notin C_i} (e^{-Dist_q(q_i, R_x)} - e^{-Dist_e(q_i, p_i^{\lambda})}) \\
&\leq 0
\end{aligned}$$

Thus, for any candidate R_x within C, we have $Sim(Q, R_x) \leq UB(R_x)$. Algorithm 2 shows the refinement process.

Algorithm 1: IKNN()

```
   input : $k$, $Q$
   output: $k$-BCT
1  Candidate Set $C$;
2  Upperbound $UB_n$;
3  Lowerbounds $LB[]$, $k\text{-}LB[]$;
4  Integer $\lambda \leftarrow k$;
5  while true do
6      for each $q_i \in Q$ from $q_1$ to $q_m$ do
7          $\lambda$-NN$(q_i) \leftarrow$ KNN$(q_i, \lambda)$;
8          $C_i \leftarrow$ trajectories scanned by $\lambda$-NN$(q_i)$;
9      $C \leftarrow C_1 \cup C_2 \cdots \cup C_m$;
10     if $|C| \geq k$ then
11         compute $LB[]$ for all trajectories in $C$;
12         compute $UB_n$;
13         $k\text{-}LB[] \leftarrow LB[]$.topK();
14         if $k\text{-}LB[].min \geq UB_n$ then
15             $k$-BCT $\leftarrow$ refine($C$);
16             return $k$-BCT;
17     $\lambda \leftarrow \lambda + \Delta$;
```

Algorithm 2: refine(Candidate Set C)

```
1  $k$-BCT $\leftarrow$ SortedList($k$);
2  compute $UB$ for each candidate in $C$;
3  sort candidates in $C$ by $UB$ in descending order;
4  for $x = 1$ to $|C|$ do
5      compute $Sim(Q, R_x)$ by traversing $R_x$;
6      if $x \leq k$ then $k$-BCT.insert($R_x$, $Sim(Q, R_x)$);
7      else
8          if $Sim(Q, R_x) > k$-BCT.$min$ then
9              $k$-BCT.removeLast();
10             $k$-BCT.insert($R_x$, $Sim(Q, R_x)$);
11         if $x = |C|$ or $k$-BCT.$min \geq UB(R_{x+1})$ then
12             return $k$-BCT;
```

In Algorithm 2, the k-BCT maintains a list of k best-connected trajectories found so far, together with the exact similarity. As the algorithm examines the candidates from C in the descending order of UB, once the current minimum similarity of the k-BCT is even larger than or equal to the UB of the next candidate R_{x+1} (line 11), the k-BCT is safely returned as the final result. Here the exact similarity of a trajectory is computed by traversing all the points on it through the double linked list (line 5), and the k-BCT is updated at line 8-10 whenever a more similar trajectory is discovered.

4.2 Adaption of the λ-NN Algorithm

The λ-NN search is a major component of the IKNN as it determines the efficiency of retrieving the λ nearest trajectory points. Basically, we have two options in designing the λ-NN function, i.e., KNN(q_i, λ), invoked at line 7 of the Algorithm 1. The first one is the *best-first* k-NN algorithm in [14], and the second choice is the *depth-first* k-NN approach in [19]. They are tree traversal methods and apply to R-tree index. The *best-first* approach, although provides an I/O optimal solution, has a risk in huge memory usage, while the *depth-first* one does have a guarantee of low memory usage but is sub-optimal in query processing. In the following, we elaborate how to adapt them to the IKNN separately, and how to significantly improve the performance of the *depth-first* strategy. Specifically, we use existing cost models [3, 15] to estimate the cost of the *best-first* strategy, and develop a new upper bound of cost for the *depth-first* strategy.

4.2.1 The Best-First Strategy

The *best-first* approach in [14], maintains a priority queue to store all the R-tree entries that have yet to be visited, using the MINDIST from the query location to an MBR as a key. Initially, the queue only contains the entries of the root. When deciding which entry to traverse next, it picks the entry with the least MINDIST (i.e. head of the queue). After the selected entry is visited, it is removed from the queue and it's children entries are subsequently put on the queue. In such a manner, an entry is not examined until it reaches the head of the queue. Therefore, the first data point that comes to the head must be the 1^{st} nearest neighbor since all entries closer to the query location have already been examined, and then the data point is dequeued. As this process proceeds, the next nearest neighbor is reported incrementally whenever it reaches the head of the queue.

Adopting this algorithm, we can simply record the priority queue of each λ-NN search, and resume the search at a later time for the $(\lambda + \Delta)$-NN by using the previous priority queue. Thus, each time the IKNN further requires Δ more trajectory points because the Theorem 1 has not been met, the KNN(q_i, λ) function in Algorithm 1 can restart the traversal from the previous queue to get the $(\lambda+1)^{th}, (\lambda+2)^{th}, \cdots, (\lambda+\Delta)^{th}$ nearest points. The advantage is that there is no revisit to any processed entry and each λ-NN point is examined only once. We call the IKNN using this *best-first* strategy the *IKNN$_{bf}$ Algorithm.*

The complexity of the IKNN$_{bf}$ is obviously the cost of the *best-first* λ-NN search multiplying a constant m. Here we borrow some formulas from existing work to show the R-tree node access when a k-BCT query is returned. Specifically, we estimate the total leaf access when the m query locations all involve a λ-NN search. Firstly, given N points in an E-dimensional unit space, the average number of neighbors $\overline{nb}(\epsilon, \text{`shape'})$ of a point within a region of regular 'shape' and radius ϵ is given by [3]:

$$\overline{nb}(\epsilon, \text{`shape'}) = \left(\frac{vol(\epsilon, \text{`shape'})}{vol(\epsilon, \Box)}\right)^{\frac{D_2}{E}} \times (N-1) \times (2\epsilon)^{D_2} \quad (6)$$

D_2 is the Correlation Fractal Dimension [3] and $vol(\epsilon, \text{`shape'})$ indicates the volume of a shape of radius $\epsilon$. In a two-dimensional space ($E = 2$), $vol(\epsilon, \Box)$ is the volume of a square, and firstly we want to estimate the radius $r$ of a circle that encloses $\lambda$ trajectory points. Since $vol(r, \text{`circle'}) = \pi r^2$ and $vol(r, \Box) = (2r)^2$, Equation 6 is transformed to:

$$\begin{aligned}\overline{nb}(r, \text{`circle'}) &= (\tfrac{\pi r^2}{4r^2})^{\frac{D_2}{2}} \times (N-1) \times (2r)^{D_2} \\ &= (N-1)(r\sqrt{\pi})^{D_2}\end{aligned} \quad (7)$$

By substituting $\overline{nb}(r, \text{`circle'})$ with λ and some simple algebraic manipulations, we get the radius r of a λ-NN search region in Equation 8, where r is actually the distance from the λ^{th} NN to the query location.

$$r = \frac{1}{\sqrt{\pi}} \sqrt[D_2]{\frac{\lambda}{N-1}} \quad (8)$$

For a *best-first* λ-NN search, only MBRs that intersected by the circle of radius r are visited. Hence it turns into a specific range query problem asking the number of node accesses w.r.t a search circle of radius r. To estimate it, we adopt the formulas in [15]. Firstly, assume the average side of MBR at level j of the R-tree is σ_j, and σ_j is given using the Hausdorff Dimension D_0 [15].

$$\sigma_j = (\frac{c^{h-j}}{N})^{\frac{1}{D_0}} \quad (9)$$

where $h = \log_c(N)$ is the height of the R-tree and c is the fanout. Then the average node access $P(\lambda)$ of R-tree for answering a range query with radius r is given by [15]:

$$P(\lambda) = \sum_{j=0}^{h-1} \frac{(N-1)(\sigma_j^2 + 4r\sigma_j + \pi r^2)^{\frac{D_2}{2}} + 1}{c^{h-j}} \quad (10)$$

The idea of Equation 10 is to sum up the access probability of every single node from the root ($j = 0$) to the leaf level ($j = h - 1$). For simplicity, we just take the node access $P_{leaf}(\lambda)$ of leaves into consideration ($j = h - 1$ only), and assume a uniform distribution of trajectory points (i.e. $D_0 = D_2 = 2$, $\sigma_{h-1} = \sqrt{c/N}$). Therefore, by simplifying Equation 10, we get the leaf access of a *best-first* λ-NN search:

$$\begin{aligned}P_{leaf}(\lambda) &= \frac{(N-1)(\sigma_{h-1}^2 + 4r\sigma_{h-1} + \pi r^2)^{\frac{D_2}{2}} + 1}{c} \\ &= \frac{N-1}{N} + 4\sqrt{\frac{\lambda(N-1)}{\pi c N}} + \frac{\lambda+1}{c} \\ &\approx 1 + 4\sqrt{\frac{\lambda}{\pi c}} + \frac{\lambda+1}{c} = O(\sqrt{\lambda} + \lambda)\end{aligned} \quad (11)$$

For the IKNN$_{bf}$ algorithm, there is no overlap between the search region of the λ-NN and the region of the $(\lambda+\Delta)$-NN. Therefore, the total leaf access for IKNN$_{bf}$ is:

$$LeafAccess_{bf} = m \times O(\sqrt{\lambda} + \lambda) \quad (12)$$

However, one major concern here is that the size of the priority queue used by the IKNN$_{bf}$ could potentially become very large. In an extreme case, it has to keep all the entries of the R-tree in the queue [14], which could easily occupy the main memory and possibly bring down the system. Consequently, no guarantee of memory usage is given by the IKNN$_{bf}$, and that is why we introduce the second option: the *depth-first* strategy, which can theoretically guarantee the usage of memory.

4.2.2 The Depth-First Strategy

In the *depth-first* k-NN algorithm [19], the basic idea is to recursively traverse the R-tree level by level in a depth-first manner, while maintaining a *global list* of k nearest candidates. Therefore, starting from the root of the R-tree, the entries of an internal node (organized as a BranchList) are sorted according to the MINDIST from the query location, and the entry with the smallest MINDIST is visited first. This downward process repeats recursively until reaching a leaf node where a potential nearest neighbor may be found. During the backtracking to upper levels, it only visits the entries whose MINDIST is smaller than the distance of the k^{th} nearest candidate in the global list found so far. With this method, we need a recursion stack to keep track of what entries have yet to be visited. There are totally $\log_c(N)$ levels, and the BranchList at each level has a size of c, where c is the fanout of the R-tree, so the size of the stack is ascertained to be $O(c \log_c(N))$. Here, we mention the IKNN using this *depth-first* strategy the *IKNN$_{df}$ Algorithm*.

Nevertheless, the price paid for fixed memory usage by this method is that we can no longer simply record the state of the traversal where the λ-NN search stopped previously and resume it later for the $(\lambda + \Delta)$-NN, because some valid candidates for the $(\lambda+\Delta)$-NN may have been skipped during the previous search. As a result, we have to restart the search from the very beginning and the previous λ-NN points will be re-scanned by the $(\lambda + \Delta)$-NN search. Assume λ is initially k and $\Delta = k$ as well for ease of proof. When a *depth-first* λ-NN search returns, each of the $1^{st}, 2^{nd}, \cdots, k^{th}$ nearest points has been visited by $\frac{\lambda}{k}$ times, and each of the $(k+1)^{th}, (k+2)^{th}, \cdots, (2k)^{th}$ nearest points has been visited by $\frac{\lambda}{k} - 1$ times, and so on. Thus, the total number of visits V_{sum} of all the λ-NN points is:

$$\begin{aligned} V_{sum} &= \textstyle\sum_{i=1}^{\frac{\lambda}{k}} (k \times i) = k + 2k + 3k + \cdots + \frac{\lambda}{k}k \\ &= \frac{k}{2}(\frac{\lambda}{k} + (\frac{\lambda}{k})^2) = O(\frac{\lambda^2}{k}) \end{aligned}$$

In the case that $\lambda \gg k$, the performance degrades dramatically, and indeed this phenomenon happens frequently in a large dataset which is point-intensive as we need to search for a comparatively large λ before determining the k-BCT. In contrast, the V_{sum} of the *best-first* strategy is always λ.

Intuitively if we can guess correctly how large λ will eventually be, this *depth-first* algorithm can go directly for the λ-NN without any repetition of search. However, this is nearly impossible even though the trajectory distribution is known in advance. To save cost, we suppose to reduce the number of repetitions. Here, instead of increasing λ by a constant Δ, we double λ at each round, that is, $\lambda = k \times 2^{round}$, with $round = 0, 1, 2, \cdots$, and hence $\Delta = k \times 2^{round}$. By doing so, the V_{sum} of all the λ-NN points is reduced to:

$$\begin{aligned} V_{sum} &= \textstyle\sum_{i=0}^{\log_2(\frac{\lambda}{k})} (k \times 2^i) = k(2^0 + 2^1 + \cdots + 2^{\log_2(\frac{\lambda}{k})}) \\ &= 2\lambda - k = O(\lambda) \end{aligned}$$

This is similar to allocating space for dynamic tables. Theoretically, the V_{sum} drops from quadratic to linear, although λ may be over larger than necessary. As a consequence, now we are able to use the *depth-first* λ-NN algorithm directly for the IKNN$_{df}$ with the line 17 in Algorithm 1 changed to '$\lambda \leftarrow \lambda \times 2$'. Although it is a very simply modification, it brings down V_{sum} to the same order of magnitude as that of the *best-first* strategy.

Practically, we can further improve the search efficiency by using the MAXDIST that is defined as the distance from the query location q_i to the furthest position on a MBR as illustrated in Figure 3.

$$\text{MAXDIST}(q_i, \text{MBR}) = \max_{p_j \in \text{MBR}} \{Dist_e(q_i, p_j)\} \tag{13}$$

By using MAXDIST, when the *depth-first* algorithm is traversing the R-tree for the $(\lambda + \Delta)$-NN, all encountered MBRs whose MAXDIST is smaller than the distance $Dist_{last}$ of the λ^{th} NN found in the previous round can be safely pruned, since all the points they contain must have been processed already. As exemplified in Figure 3, MBR1 and MBR2 can be skipped since they are totally covered by the circle with center q_i and radius $Dist_{last}$. However, MBRs intersect with the circle partially, e.g., MBR3 and MBR4, still need to be processed even though they were visited before. The modified *depth-first* traversal method is shown in Algorithm 3 which is simplified from the one in [19] for an easier illustration. Here the only change is at line 6-7, where it skips the MBR contained by the circle of radius $Dist_{last}$.

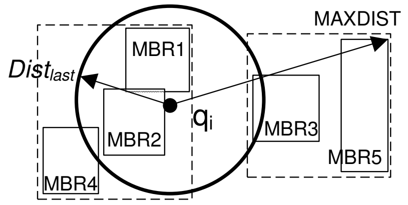

Figure 3: Pruning by MAXDIST

Algorithm 3: DepthTraversal($Node$)

```
1  if $Node$ = $Leaf$ then update $result$;
2  else
3      $BranchList \leftarrow Node.entries$;
4      sort $BranchList$ by MINDIST;
5      for each $e \in BranchList$ do
6          if MAXDIST($q_i$, $e$.MBR) $< Dist_{last}$ then
7              skip $e$;
8          else if MINDIST($q_i$, $e$.MBR) $< result.max$ then
9              DepthTraversal($e$);
10         else break;
```

To estimate the leaf access of the IKNN$_{df}$, however, we can not use Equation 11 directly, since in a *depth-first* algorithm probably some leaf nodes that are further than the λ^{th} NN will be visited as well. Due to the difficulty in estimating it's exact cost [4], here we just derive an upper bound of leaf access. Note that an upper bound for 1-NN search is given in [17], but it is not applicable for the λ-NN case.

In our derivation, denote by MBR$_1$ the first visited MBR (at any level) that contains not less than λ points during the *depth-first* traversal (MBR$_1$ is not necessarily to center at the query location). Obviously, a full global list with λ candidates is generated after visiting MBR$_1$, and any MBR whose MINDIST is larger than the distance of the *furthest point* in the global list will be pruned. Assume r_{max} is the distance from the furthest point to the query location as illustrated in Figure 4. Only MBRs intersected by the circle with center q_i and radius r_{max} can be visited after MBR$_1$ is examined. Therefore, we can estimate an upper bound of the exact leaf access by using the leaf access w.r.t. the search circle of radius r_{max}. Then Equation 11 can be used directly for calculation by setting $r = r_{max}$.

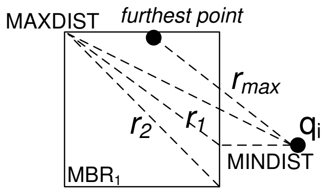

Figure 4: Estimating MAXDIST(q_i, MBR_1)

Although r_{max} is not easy to be determined, we can upperbound r_{max} by $r_{max} \leq$MAXDIST(q_i, MBR_1) because the furthest point is enclosed by MBR$_1$. Thus, now the problem is how to estimate MAXDIST(q_i, MBR_1). As shown in Figure 4, denote by r_1 the distance between the closest position and the furthest position to q_i on the boundary of MBR$_1$, and by r_2 the length of the diagonal of MBR$_1$ (obviously $r_2 \geq r_1$). According to triangle inequality, we have:

$$\begin{aligned} \mathrm{MAXDIST}(q_i, \mathrm{MBR}_1) &\leq r_1 + \mathrm{MINDIST}(q_i, \mathrm{MBR}_1) \\ &\leq r_2 + \mathrm{MINDIST}(q_i, \mathrm{MBR}_1) \end{aligned} \quad (14)$$

We further notice that:

$$\mathrm{MINDIST}(q_i, \mathrm{MBR}_1) \leq Dist_e(q_i, 1^{st}\mathrm{NN}) \quad (15)$$

The proof of Inequation 15 is as follows: *If the 1^{st} nearest point of q_i locates within MBR$_1$, then MINDIST$(q_i, MBR_1) \leq Dist_e(q_i, 1^{st}NN)$ as MINDIST is the shortest distance to q_i. Otherwise, if the 1^{st} NN is enclosed by some other $MBR^{'}$ at the same level, then we must have MINDIST$(q_i, MBR^{'}) \leq Dist_e(q_i, 1^{st}NN)$. Since MBR$_1$ is the first visited MBR that has the minimum MINDIST among all the MBRs at it's level, we have MINDIST$(q_i, MBR_1) \leq$ MINDIST$(q_i, MBR^{'}) \leq Dist_e(q_i, 1^{st}NN)$. Therefore, Inequation 15 is proved.*

Combining Formula 14 and Formula 15, MAXDIST(q_i, MBR_1) is further upperbounded as follows:

$$\mathrm{MAXDIST}(q_i, \mathrm{MBR}_1) \leq r_2 + Dist_e(q_i, 1^{st}\mathrm{NN}) \quad (16)$$

In the following, we reckon r_2 and the distance of the 1^{st} NN. Considering Equation 8, we set $\lambda = 1$ and get:

$$Dist_e(q_i, 1^{st}\mathrm{NN}) = \frac{1}{\sqrt{\pi}} \sqrt[D_2]{\frac{1}{N-1}} \quad (17)$$

For estimating r_2, again, we use Equation 6 by setting 'shape = MBR'. Here, we take the same assumption from [17] that all MBRs are squares with side 2ϵ. Let $\overline{nb}(\epsilon, \text{'MBR'}) = \lambda$. The radius of a MBR that contains λ points is estimated as:

$$\epsilon = \frac{1}{2} \sqrt[D_2]{\frac{\lambda}{N-1}} \quad (18)$$

We have $r_2 = 2\sqrt{2}\epsilon$, $D_2 = 2$ and consequently:

$$\mathrm{MAXDIST}(q_i, \mathrm{MBR}_1) \leq \sqrt{\frac{2\lambda}{N-1}} + \sqrt{\frac{1}{\pi(N-1)}} \quad (19)$$

Let r = MAXDIST(q_i, MBR_1) in Equation 11. Through similar algebraic manipulations, we finally estimate an upper bound of leaf access for the *depth-first* λ-NN search by:

$$\begin{aligned} P_{leaf}(\lambda) &\approx (4 + 2\sqrt{\pi})\sqrt{\tfrac{2\lambda}{c}} + \tfrac{2\pi\lambda+2}{c} + \sqrt{\tfrac{1}{\pi c}} + 1 \\ &= O(\sqrt{\lambda} + \lambda) \end{aligned} \quad (20)$$

In a *depth-first* traversal, the actual leaf access should be smaller than the $P_{leaf}(\lambda)$ in Equation 20, since not all the leaves within range r need to be visited with the global list being updated during the search. From this analysis, we can also see that both the *best-first* and *depth-first* λ-NN search involve the same order of magnitude of leaf access.

For the IKNN$_{df}$ strategy, the $(\lambda + \Delta)$-NN will re-scan the search region of the λ-NN more or less even though MAXDIST is used for pruning. However, the order of complexity is expected to be unchanged, if we double λ at each round.

$$\begin{aligned} LeafAccess_{df} &\leq m \times \textstyle\sum_{i=0}^{\log_2(\frac{\lambda}{k})} P_{leaf}(k \times 2^i) \\ &= m \times O(\sqrt{\lambda} + \lambda) \end{aligned} \quad (21)$$

As a result, we prove that the cost of the IKNN$_{df}$ and IKNN$_{bf}$ tend to be similar through doubling λ, while the IKNN$_{df}$ also avoids potential high memory usage that can not be guaranteed by the IKNN$_{bf}$. However, it does not mean that the IKNN$_{bf}$ is worse. In a normal case, the priority queue can still be accommodated in main memory easily and typically the IKNN$_{bf}$ is faster than the IKNN$_{df}$ as it involves no repetition.

4.3 Optimization

In the previous setting of the IKNN algorithm, query locations are treated equally by increasing λ by the same Δ for each λ-NN(q_i). However, note that not all the query locations are of equal importance as different λ-NN(q_i) probably have different contributions in constructing the candidate set and determining lower/upper bounds of similarity. For instance, given two query locations q_i and q_j, if $Dist_e(q_i, p_i^{\lambda}) > Dist_e(q_j, p_j^{\lambda})$ for the same λ, which means the λ^{th} NN of q_i is farther, then $e^{-Dist_e(q_i,p_i^{\lambda})}$ is smaller than $e^{-Dist_e(q_j,p_j^{\lambda})}$. Since $UB_n = e^{-Dist_e(q_1,p_1^{\lambda})} + e^{-Dist_e(q_2,p_2^{\lambda})} + \cdots + e^{-Dist_e(q_m,p_m^{\lambda})}$ according to Formula 4, we may say the λ^{th} NN of q_i helps more than the λ^{th} NN of q_j does in lowering the UB_n. Apparently, the lower the UB_n is, the easier Theorem 1 would be satisfied, and thus the IKNN would return results more quickly. In the following, we analyze how the contribution of q_i is affected by λ. Firstly, we define the contribution of q_i to UB_n by $\xi(q_i)$:

$$\xi(q_i) = e^{-Dist_e(q_i,p_i^{\lambda})}$$

Obviously the smaller contribution q_1 to q_m holds, the smaller UB_n will be. Denote the density of trajectory points by ρ and the radius of a λ-NN search by $r = Dist_e(q_i, p_i^{\lambda})$. Within the region of a λ-NN search, we roughly estimate ρ as

$$\rho = \frac{\lambda}{\pi r^2}$$

Easily, we have $r = \sqrt{\frac{\lambda}{\pi\rho}}$ and $\xi(q_i)$ is rewritten as

$$\xi(q_i) = e^{-r} = e^{-\sqrt{\frac{\lambda}{\pi\rho}}}$$

Here, our target is to figure out how fast $\xi(q_i)$ decays as λ increases, and then assign different Δ for q_1 to q_m according to it's decay rate when proceeding to the $(\lambda+\Delta)$-NN search. We take the derivative $\frac{d\xi}{d\lambda}$ of $\xi(q_i)$ as the *decay rate* $Dec(q_i)$:

$$\frac{d\xi}{d\lambda} = \frac{d}{d\lambda}e^{-\sqrt{\frac{\lambda}{\pi\rho}}} = -\frac{1}{2}(\pi\rho\lambda)^{-\frac{1}{2}} \times e^{-\sqrt{\frac{\lambda}{\pi\rho}}} \quad (22)$$

Each time of calculating $\frac{d\xi}{d\lambda}$, again, we approximate ρ in Equation 22 by $\frac{\lambda}{\pi r^2}$ using the current λ and r. Thus, the current decay rate is approximated by:

$$Dec(q_i) = |\frac{d\xi}{d\lambda}| = \frac{r}{2\lambda}e^{-r} \quad (23)$$

We can see that, given a fixed λ, the decay rate rises first and then drops gradually as r grows from 0 to ∞. Thus, in the beginning we should assign a higher priority in exploring a query location with a sparser distribution of trajectories (i.e. with a larger r). However, after r reaches some value, instead, a denser distribution (i.e. with a comparatively smaller r) generates a larger decay rate. Therefore, more effort should be made for query locations with dense trajectories around. By doing so, we can decrease the UB_n more efficiently. Nevertheless, when r and λ are all large enough, a further search for more trajectory points doesn't help any more in reducing UB_n because $\frac{d\xi}{d\lambda}$ is already close to 0.

Another potential way to accelerate the IKNN algorithm is to increase the lower bound LB of candidates as fast as possible, because larger lower bounds also make Theorem 1 easier to be satisfied. However, the lower bound of a trajectory is derived from the search results of more than one query locations, and it is not easy to predict when and where a λ-NN(q_i) will meet with a point of a given trajectory. Consequently, regarding LB, we face difficulties in quantitatively estimating the contribution of a query location. As an alternative solution, we define the *retrieval ratio* $Rat(q_i)$ as a search heuristic, based on the number of new trajectories found by the λ-NN(q_i) at each round.

$$Rat(q_i) = \frac{Num(q_i)}{\lambda - \lambda'} \tag{24}$$

where λ' indicates the λ of the previous round, and $Num(q_i)$ is the number of new trajectories found by the λ-NN. The idea is that the more new trajectories are retrieved, the larger candidate set C will be, and it is likely to discover more points on the k best-connected trajectories. Thus the lower bounds probably rise more quickly.

By considering both the decay rate and retrieval ratio, we increase Δ for different query locations accordingly.

$$\Delta(q_i) = \delta\left(\mu\frac{Dec(q_i)}{\sum_{i=1}^{m} Dec(q_i)} + \nu\frac{Rat(q_i)}{\sum_{i=1}^{m} Rat(q_i)}\right) \tag{25}$$

In Equation 25, μ and ν are weights, and $\delta = m \cdot k \cdot 2^{round}$ for the IKNN$_{df}$, while δ is equal to m multiplying some constant Δ for the IKNN$_{bf}$. As a result, instead of increasing λ for all query locations equally at line 17 of Algorithm 1, the optimized IKNN determines λ for each q_i separately according to Equation 25 (i.e. $\lambda(q_i) = \lambda(q_i) + \Delta(q_i)$), and therefore more effort is put on exploring important query locations which may potentially accelerates the searching of the k-BCT. Note that the optimized IKNN explores totally $\sum_{i=1}^{m}\Delta(q_i)$ NN points at each round, and we have:

$$\begin{aligned}&\textstyle\sum_{i=1}^{m}\Delta(q_i) = \delta\left(\mu\frac{\sum_{i=1}^{m} Dec(q_i)}{\sum_{i=1}^{m} Dec(q_i)} + \nu\frac{\sum_{i=1}^{m} Rat(q_i)}{\sum_{i=1}^{m} Rat(q_i)}\right)\\ &= \delta\cdot(\mu+\nu)\end{aligned}$$

To guarantee the IKNN searches for a certain number of points per round, we set $\mu + \nu = 1$ (e.g. $\mu = \nu = 0.5$), so the total number of new points got per round is always δ.

4.4 Extension to Queries with an Order

Provided the query is with an order constraint, as discussed in Section 3, we further consider the visiting order in the matching between query locations and trajectories. In this case, the similarity is measured by Sim_o as formulated in Equation 2. Typically, Sim_o can be solved by a Dynamic Programming (DP) paradigm as shown in Algorithm 4, which conducts a shifting on trajectory $R = \{p_1, p_2, \ldots, p_l\}$ by repeating/skipping some p_i in order to best-match with the query, and finally figures out $Sim_o(Q, R)$.

Algorithm 4: $DP(Q, R)$

```
1  Matrix M[i,j];
2  ∀i ∈ [1, m], M[i,0] ← 0;
3  ∀j ∈ [1, l], M[0,j] ← 0;
4  for i = 1 to m do
5      for j = 1 to l do
6          if $e^{-Dist_e(Head(Q),Head(R))}$+M[i-1,j] > M[i,j-1] then
               // match $q_i$ with $p_j$ and repeat $p_j$
7              M[i,j] ← $e^{-Dist_e(Head(Q),Head(R))}$+M[i − 1,j];
8          else
               // skip $p_j$
9              M[i,j] ← M[i,j − 1];
10 return M[m,l];
```

Here M[i,j] records the similarity of a subproblem, i.e., $Sim_o(\{q_1, \cdots, q_i\}, \{p_1, \cdots, p_j\})$. Once we got M[$i-1$,$j$] and M[$i$,$j-1$], we compare $e^{-Dist_e(Head(Q),Head(R))}$+M[$i-1$, j] with M[i, $j-1$] at line 6 and take the maximum as M[i,j]. If the former one is larger, we say q_i is matched with p_j and accumulate $e^{-Dist_e(Head(Q),Head(R))}$ to Sim_o, otherwise p_j is skipped. In such a bottom-up manner, Algorithm 4 figures out M[i,j] with i goes from 1 to m and j from 1 to l, where m is number of query locations and l is the number of trajectory points. Finally the M[m,l] returned by $DP(Q, R)$ at line 10 is the exact $Sim_o(Q, R)$. The complexity of the algorithm is $O(l \cdot m)$ and it tends to be linear since m is a small constant.

Now the problem is how to adapt the IKNN algorithm to find the k-BCT with respect to Sim_o, observing that the lower bound LB and upper bound UB are not applicable any more as they are designed for Sim. Firstly, for a candidate trajectory $R_x \in C$ generated by the IKNN, some of it's trajectory points are scanned by the λ-NN search. Denote the set of scanned points on R_x by R_x', and we get:

$$R_x' = \{p_i | p_i \in R_x \wedge p_i \in S\}$$

where $S = \lambda\text{-NN}(q_1) \cup \lambda\text{-NN}(q_2) \cup \cdots \cup \lambda\text{-NN}(q_m)$. Actually R_x' is a sub-trajectory that includes only a subset of points on R_x. Let R_x' still follow the order of R_x. we have:

$$Sim_o(Q, R_x) \geq Sim_o(Q, R_x') \tag{26}$$

The proof of Inequation 26 is as follows: *Suppose on the contrary there exists a matching between Q and R_x' with $Sim_o(Q, R_x') > Sim_o(Q, R_x)$. Assume the matched pairs between Q and R_x' are $\{< q_1, p(w_1) >, < q_2, p(w_2) >, \cdots, < q_m, p(w_m) >\}$ where $\{p(w_1), p(w_2), \cdots, p(w_m)\}$ are the matched points on R_x', and $Sim_o(Q, R_x') = \sum_{i=1}^{m} e^{-Dist_e(q_i, p(w_i))}$. Since $R_x' \subseteq R_x$, $\{p(w_1), p(w_2), \cdots, p(w_m)\}$ must be contained in R_x. Therefore the above matching can be applied on R_x as well and consequently $Sim_o(Q, R_x)$ should be at least $Sim_o(Q, R_x')$, which is a contradiction with the assumption.*

Based on Inequation 26, we define a new lower bound LB_o of similarity for ordered query locations by using the partially retrieved trajectory points of R_x.

$$LB_o(R_x) = Sim_o(Q, R_x') = DP(Q, R_x') \tag{27}$$

where $DP(Q, R_x')$ is calculated using Algorithm 4. For the refinement step, similar to the definition of UB in Equation 5, we define a new upper bound UB_o for candidate trajectories within the candidate set C only.

$$UB_o(R_x) = LB_o(R_x) + \sum_{i \in [1,m] \wedge R_x \notin C_i} (e^{-Dist_e(q_i, p_i^{\lambda})}) \quad (28)$$

Therefore, we replace the LB in Algorithm 1 and the UB in Algorithm 2 with LB_o and UB_o respectively, and then the IKNN algorithm can be seamlessly adapted to k-BCT queries with an order constraint.

5. EXPERIMENTS

In this section, we conduct experiments on the Beijing dataset which consists of 12,653 GPS trajectories (1,147,116 points) collected by the Microsoft GeoLife Project [1], and the distribution of the dataset is illustrated in Figure 5. The IKNN algorithm is implemented in Java and examined on a windows platform with Intel Core 2 CPU (2.13GHz) and 1.0GB Memory. All the trajectory points are indexed by one single R-tree. In the experiments, k-BCT queries are collected manually by selecting a sequence of coordinates of places of interest that complies with a reasonable visiting order. We don't use random generation for query locations as it may cause a sudden jump from one location to another far-away location that probably won't happen in real life.

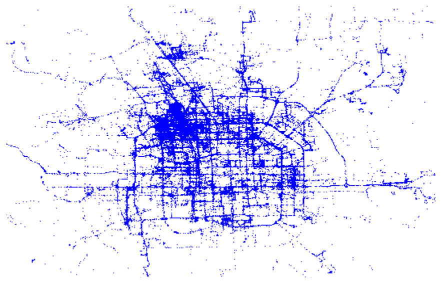

Figure 5: The Beijing dataset

The main metric we adopt for measuring the performance is the *Query Time* that reflects how fast a query is returned, and the *Node Access* that indicates the number of visits to R-tree nodes. Here we simply record how many times the internal nodes and leaf nodes of the R-tree are accessed during a query. Besides, we also compare the memory usage by using the *Queue Size* which is the number of elements of the m priority queues in the IKNN$_{bf}$ algorithm, or the total size of the branch lists in the IKNN$_{df}$ algorithm. As this is the first work on searching trajectory by locations, we only compare the performance of the IKNN with different settings. The IKNN$_{bf}$ with the *best-frist* strategy is denoted by BF, while the IKNN$_{bf}$ with optimization is mentioned as BF-O. Similarly, the IKNN$_{df}$ that uses a constant Δ is indicated by DF-C, and DF-D is referred to as the IKNN$_{df}$ that doubles λ at each round. DF-D-M further considers MAXDIST for pruning, and DF-D-M-O also includes the optimization mechanism. The experiment setting is as follows:

the number of k :	1 to 25, default 15
the number of locations :	2 to 10, default 8
the constant Δ *for BF and DF-C* :	50
the μ, ν *for optimization* :	0.5

Note that adjusting Δ, μ and ν also affects the performance perceptibly although the trend is still similar. The figures are not shown due to the limit of space.

5.1 Different Number of Query Locations

The number of query locations is a critical assumption in our applications, in which we assume the number is small as it's impractical to input tens of locations for a query and importantly a few locations are enough to search a trajectory around some places of interest. In this part, we fix k to 15 and compare the IKNN variants as the number of query locations ranges from 2 to 10. In effect, the algorithm works without any problem even though the number of query locations goes up to 100 or more, but in this case the problem itself is no longer very meaningful in our applications.

5.1.1 Best-first vs. Depth-first

As shown in Figure 6(a) and 6(b), the *best-first* method BF notably outperforms the *depth-first* method DF-C without any optimization. As the number of query locations goes up to 10, the *Query Time* and *Node Access* of the DF-C rise quickly to more than 800 ms and $14,000$ respectively, while the BF just raises the cost smoothly with *Query Time*$\approx$ 200 ms and *Node Access*$\approx$ 800. The root cause is that the DF-C involves an excessive number of re-visits to previous λ-NN search regions and consequently produces a much larger number of *Node Access* compared with the BF that intrinsically avoids any overlap of search regions. However, the price paid by the BF to achieve high performance is high memory usage. As shown in Figure 6(c), the *Queue Size* of the BF is approximately 10 times larger than that of all the *depth-first* variants. With a very skewed dataset distribution, it is believed that much more queue capacity is required by the BF. Nevertheless, even though the *Queue Size* peaks to above 700, it still can be fitted in main memory easily (but a smaller number of concurrent queries). Therefore, neglecting very bad cases, the *best-first* strategy is the best choice for processing the k-BCT queries.

5.1.2 Effect of Adaption

To achieve a guarantee of low memory usage, we can consider the *depth-first* strategy. However, as observed, the DF-C doesn't scale well and further adaption is necessary. Firstly, we study how Δ affects the performance. As shown in Figure 6(a) and 6(b), the DF-D that doubles λ at each round presents smoother curves of the *Query Time* and *Node Access* than the DF-C does, which confirms the fact that by doubling λ, the DF-D reduces the number of rounds from $O(\lambda/\Delta)$ to $O(\log_2 \lambda)$, and thus the number of repetitions is reduced greatly. In essence, increasing λ exponentially is a fast way to achieve a necessarily large λ with a small number of rounds. Although DF-D still has to cover all the previous search regions, it already brings the search cost to the same order of magnitude as that of the BF. Pruning by MAXDIST is another adaption for the *depth-first* strategy. Again, as shown in Figure 6(a) and 6(b), DF-D-M further lowers the cost as it reduces the area of repetition remained with the DF-D. An interesting observation is that the *Node Access* of the DF-D-M drops slightly as the number of locations rises from 8 to 10. That is because more query locations may help in raising the lower bound of trajectories greatly in some cases and thus make Theorem 1 easier to be satisfied. Besides, note that the *Query Time* of the BF is even larger than that of the DF-D-M as the BF involves frequent queue operations, which increases the cost notably when the *Queue Size* is large, and the BF also has to compute lower and upper bounds for many rounds ($O(\lambda/\Delta)$).

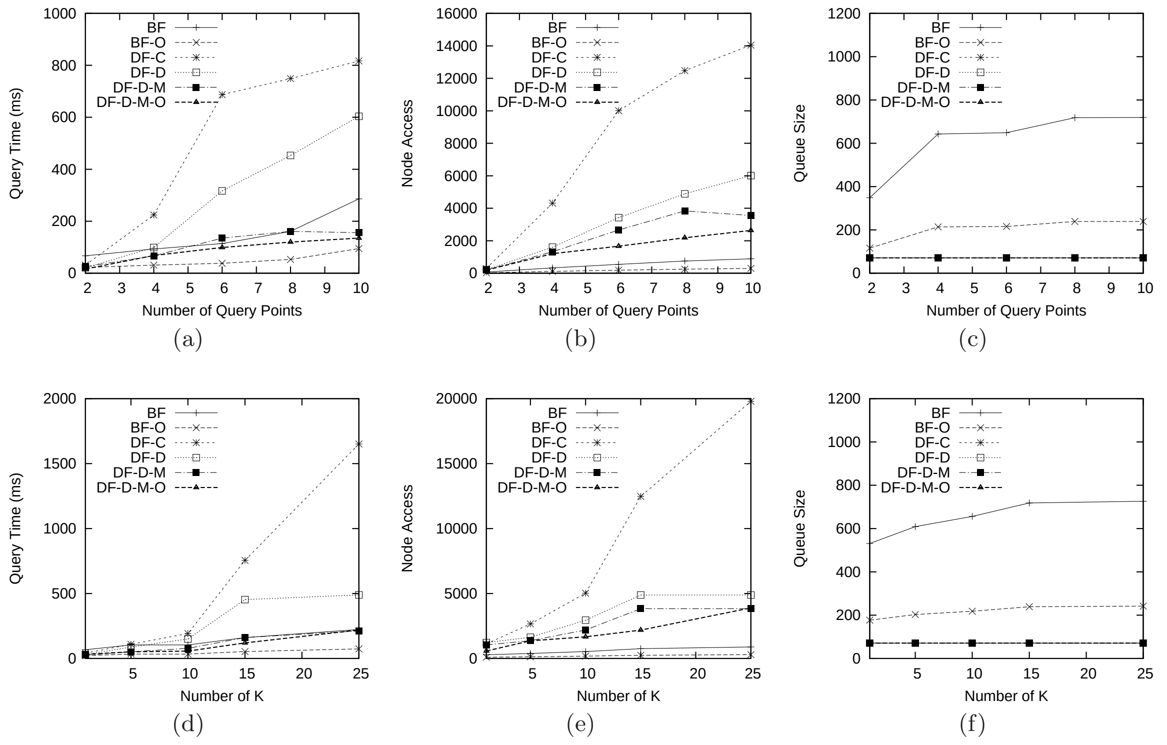

Figure 6: Performance for queries without an order

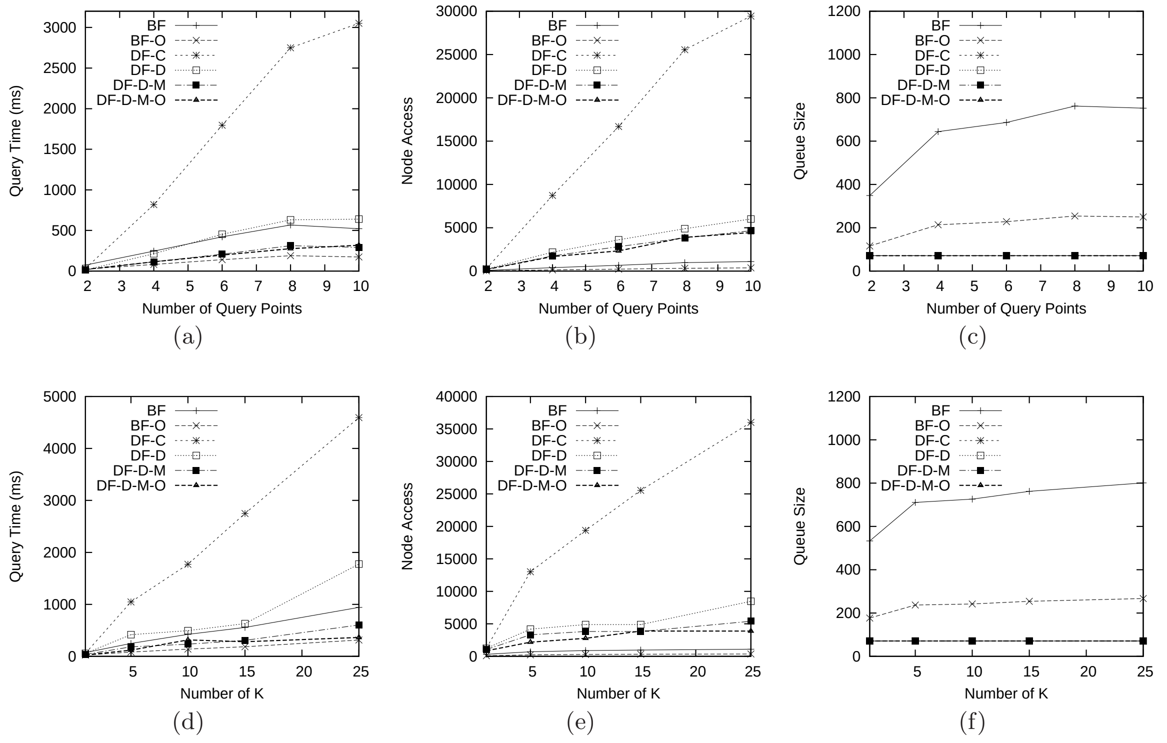

Figure 7: Performance for queries with an order

5.1.3 *Effect of Optimization*

The purpose of adjusting Δ separately for different query locations by considering their importance is to put more effort on exploring more important query locations. As expected, a further improvement over the DF-D-M is achieved by the DF-D-M-O. In Figure 6(b), the *Node Access* is further reduced by nearly $1/4 \sim 1/2$, while in Figure 6(a) the *Query Time* is reduced by about 1/5. For the *best-first* strategy, a more significant improvement is observed. The BF-O requires only 295 *Node Access* which is about 50% less than that of the BF when the number of locations is 10. Obviously, the reason of improvement is that both strategies visit fewer trajectory points during the query through optimization, and from Figure 6(c) it is seen that the *Queue Size* of the BF-O is lowered by up to 2/3 compared with the BF.

5.2 Different Number of k

Another concern about the IKNN algorithm is how it scales with different number of k. Here, we show the results in Figure 6(d), 6(e) and 6(f) with the number of locations fixed to 8. It is observed that the BF-O exposes an amazingly stable performance without any significant fluctuation in both *Query Time* and *Node Access* as k increases from 1 to 25, while the *Queue Size* is kept at a comparatively low number (≈ 200). Besides that, the DF-D-M and DF-D-M-O also appear to be scalable with k, and no significant rise in terms of *Query Time* and *Node Access* is shown. On the other hand, the figures of the DF-C boost drastically to more than 1.5 second and 20,000 respectively, which reveals that a pure *depth-first* algorithm does not work well and the adaption and optimization are very necessary.

5.3 Queries with an Order

In comparison with queries without an order, when a query is with an order constraint, more computation is needed to figure out the lower bound and the exact similarity of a trajectory by using the Algorithm 4, and furthermore a query location may not be matched with the nearest point on a trajectory any longer which consequently requires a scanning of more trajectory points to get the best matching. Therefore, more *Node Access* and *Query Time* are introduced, as confirmed in Figure 7(a), 7(b), 7(d) and 7(e) where all the variants have brought up the cost more or less. However, the trends of the figures are still similar to that of the queries without an order in Figure 6. Regarding the size of queue, it does not change perceptibly in Figure 7(c) and 7(f) compared with Figure 6(c) and 6(f).

6. CONCLUSIONS

In this paper, we study a new problem of searching the k Best-Connected Trajectories from a database by using a set of locations with or without an order constraint. Since the number of query locations is typically small, it enables us to adopt a spatial method for answering a similarity search query. We start the study based on a simple IKNN algorithm and then analyze the efficiency of different variants. As a conclusion, we would say that the BF-O achieves the best query performance although involving a risk of high memory usage. The pure DF-C algorithm, although guarantees a low memory consumption, performs poorly in efficiency. Therefore, we further devise the DF-D-M and DF-D-M-O to improve the DF-C for fewer R-tree node access and shorter query time, and finally their performance are theoretically and experimentally confirmed to be close to that of the BF.

7. REFERENCES

[1] http://research.microsoft.com/en-us/projects/geolife/.

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

[3] A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *VLDB*, pages 299–310, 1995.

[4] C. Böhm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

[5] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD*, pages 599–610, 2004.

[6] K.-P. Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

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

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

[9] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *PVLDB*, pages 1542–1552, 2008.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[11] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *Geoinformatica*, 11(2):159–193, 2007.

[12] E. Frentzos, K. Gratsias, and Y. Theodoridis. Index-based most similar trajectory search. In *ICDE*, pages 816–825, 2007.

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

[14] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[15] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *TKDE*, 13(1):96–111, 2001.

[16] M. D. Morse and J. M. Patel. An efficient and accurate method for evaluating time series similarity. In *SIGMOD*, pages 569–580, 2007.

[17] A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in r-trees. In *ICDT*, pages 394–408, 1997.

[18] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[19] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[20] R. Sherkat and D. Rafiei. On efficiently searching trajectories and archival data for historical similarities. *PVLDB*, pages 896–908, 2008.

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

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