Maximizing Reverse k-Nearest Neighbors for Trajectories

Chapter · May 2018

DOI: 10.1007/978-3-319-92013-9_21

3 authors:

Tamjid Al Rahat
University of California, Los Angeles
6 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Arif Arman
Texas A&M University
7 PUBLICATIONS **27** CITATIONS
SEE PROFILE

Mohammed Eunus Ali
Bangladesh University of Engineering and Technology
111 PUBLICATIONS **778** CITATIONS
SEE PROFILE

Maximizing Reverse k-Nearest Neighbors for Trajectories

Tamjid Al Rahat, Arif Arman, and Mohammed Eunus Ali

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
{tamjid, arman}@cse.uiu.ac.bd, eunus@cse.buet.ac.bd

Abstract. In this paper, we address a popular query involving trajectories, namely, the Maximizing Reverse k-Nearest Neighbors for Trajectories (**MaxRkNNT**) query. Given a set of existing facility trajectories (e.g., bus routes), a set of user trajectories (e.g., daily commuting routes of users) and a set of query facility trajectories (e.g., proposed new bus routes), the MaxRkNNT query finds the proposed facility trajectory that maximizes the cardinality of reverse k-Nearest Neighbors (NNs) set for the query trajectories. A major challenge in solving this problem is to deal with complex computation of nearest neighbors (or similarities) with respect to multi-point queries and data objects. To address this problem, we first introduce a generic similarity measure between a query object and a data object that helps us to define the nearest neighbors according to user requirements. Then, we propose some pruning strategies that can quickly compute k-NNs (or top-k) facility trajectories for a given user trajectory. Finally, we propose a filter and refinement technique to compute the MaxRkNNT. Our experimental results show that our proposed approach significantly outperforms the baseline for both real and synthetic datasets.

1 Introduction

With the widespread use of GPS enabled mobile devices, we have witnessed an unprecedented growth of real trajectory data (e.g., taxi or uber trajectories) capturing the daily movements of people. Such availability has enabled us to address many real life problems by developing a new range of applications based on the trajectory data. For example, let us consider a scenario where a trajectory database consists of user's daily commuting routes with their private vehicles or taxis. To improve the traffic condition and to encourage more people to use public transports, the city authority may want to introduce new facilities (e.g., bus routes) so that they can attract the maximum number of users currently using private vehicles for their regular commute. Similarly, a tourist operator may want to design its routes based on the popularity of users' visiting preferences to different points of interest (POIs). In all these applications, a user may want to avail the proposed bus service if a proposed bus route is one of k nearest bus routes (or top-k bus routes) from user trajectory. To find the best k routes for

a user trajectory, we need to define an appropriate scoring function that ranks facility trajectories with respect to the user trajectories. Moreover, this scoring function may vary across applications. Thus we first devise scoring functions that can handle a wide range of practical applications (See Section 3).

In each of the above scenarios the goal is to find a new facility trajectory that best suits for the maximum number of user trajectories. In this case, a facility trajectory is suitable for a user if the trajectory is one of the k-NN (or top-k) facilities with respect to the user trajectory. This problem can be mapped to a *reverse query* problem on a trajectory database since the underlying problem is to find the facility trajectory which is among the k-nearest neighbors (k-NN) of maximum user trajectories [1]. Therefore, in this paper we first explore the RkNN query on trajectories, which we refer to as RkNNT. Formally, we can define the RkNNT query as follows. Given a set of user trajectories $\mathcal{D}_{\mathcal{U}}$ and another set of existing facility trajectories $\mathcal{D}_{\mathcal{F}}$, an R$k$NNT query returns all users that take $q \in \mathcal{Q}$ as one of their k-nearest neighbors, where $\mathcal{Q}$ is the set of query facility trajectories.

Fig. 1 illustrates an example of the RkNNT query with two query facilities $\mathcal{Q}_1$ and $\mathcal{Q}_2$, five user trajectories $\mathcal{U}_1, \mathcal{U}_2 \ldots \mathcal{U}_5 \in \mathcal{D}_{\mathcal{U}}$, two existing facilities, $\mathcal{F}_1$ and $\mathcal{F}_2 \in \mathcal{D}_{\mathcal{F}}$ and k=1. Facility $\mathcal{F}_2$ is the 1-NN (or top-1) for users $\mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5$ and $\mathcal{F}_1$ is the NN for $\mathcal{U}_1$. When a new facility $\mathcal{Q}_2$ arrives, $\mathcal{Q}_2$ becomes the NN for $\mathcal{U}_3$, $\mathcal{U}_4$, $\mathcal{U}_5$. Thus, R1NNT($\mathcal{Q}_2$) = $\{\mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5\}$. Similarly, R1NNT($\mathcal{Q}_1$) = $\{\mathcal{U}_1\}$.

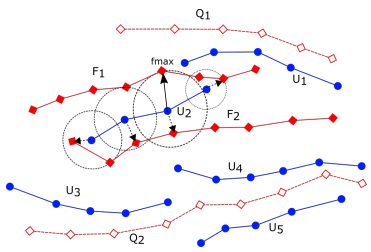


Fig. 1: An example of the **RkNNT** query, with $Q = \{\mathcal{Q}_1, \mathcal{Q}_2\}$, $U = \{\mathcal{U}_1, \mathcal{U}_2 \ldots \mathcal{U}_5\}$ and $F = \{\mathcal{F}_1, \mathcal{F}_2\}$.

The RkNNT query has been introduced in [13] recently. Though this work made an important contribution towards trajectory based reverse query processing, it has the following major limitations: (i) They have assumed that user trajectories (i.e. passenger transitions) contain only source and destination locations of the user, and provided a point-based solution where a user trajectory is considered as two separate points, (ii) They have used a simple distance function to measure the nearest neighbor from a source (or a destination) location to the

[1] In this paper we use the terms k-NN trajectories and top-k facilities interchangeably as we use different types of distances or weighted distances as a scoring function to find the best k facilities

facility trajectory. Thus, their approach is not applicable for applications that require handling other optimizations such as travel distance, priorities, etc. in the distance measure between two multi-point trajectories (user and facility).

In this paper, we first propose a class of similarity scoring functions between a multi-point user trajectory and multi-point facility trajectory that cover a wide range of practical applications. Then we propose several optimizations to compute the k nearest (top-k) facility trajectory for a given user trajectory. After that, we propose an efficient pruning technique to compute the RkNNT for all given facility queries, and return the facility that results in the maximum cardinality of RkNNT result set as the answer to the MaxRkNNT query.

In summary, the contributions of this paper are as follows.

- We define a new class of scoring functions to measure the proximity (similarity) between a user and a facility trajectory according to user preference.
- We introduce a robust algorithm to compute top-k facilities for multi-point user trajectories.
- We proposed a pruning technique to efficiently compute RkNNT, which forms the basis to answer the MaxRkNNT query.
- We conduct detailed experimental study on real datasets to show the efficiency and efficacy of the proposed solution.

2 Related Works

In this section we review some of the previous works of trajectory search and classic RkNN search.

Trajectory Search. There are several studies on single-point based query which looks for the nearest trajectories for only one static [4] or continuously moving location point [10]. Frentzos *et al.* [5] solved Nearest Neighbor Search query over the trajectories of moving object from a stationary query point. Chen *et al.* [3] proposed k Best-Connected Trajectory(k-BCT) query which searches nearest trajectory from multiple location. The similarity function used in k-BCT query does not solve our problem, because exponential function assigns larger contribution to the closer matched query point and trajectories. Shang et al. [9] proposed a Reverse Path Nearest Neighbor (R-PNN) query which finds nearest point of a given moving object dataset. There are also some studies on trajectory search for point based optimal route finding [8] and region based travel path finding [7].

The initial step in *Trajectory Search* studies is to define similarity/distance function between trajectory points. Several kinds of distance function have been defined in considerable amount of related works of trajectory search. Dynamic Time Warping (DTW) [15], Longest Common Subsequence (LCSS) [12], Edit Distance with Real Penalty (ERP) [1] and Edit Distance on Real Sequences (EDR) [2] are some of the typical distance function studied in previous works of trajectory search. However, none of these distance functions can be extended to solve our problem scenarios, where user preference is also considered along with spatial distance.

Reverse kNN. Most of existing MaxRkNN search are based on static location points that apply various pruning-refinement frameworks to avoid scanning the entire dataset. Wong et al. [14] introduced the MaxOverlap algorithm to solve MaxRkNN problem for spatial points. The algorithm iteratively finds the intersection point of the Nearest Location Circles (NLCs) that are covered by the largest number of NLCs. However, the scalability of MaxOverlap is an issue and the computation of the intersection points is also expensive. Some other existing works like MaxFirst algorithm by Zhou et al. [16], MaxSegment algorithm by Liu et al. [6] overcome the limitations of MaxOverlap. They use a variant of plane sweep to find the optimal interval. These works solely focus on static point based intersection of geometric shapes, space partitioning or sweep line techniques, and thus cannot be applied or extended to find MaxRkNN query for multi-point trajectories of variable length.

3 Problem Formulation

Formally, we define the Maximizing Reverse k-Nearest Neighbors for Trajectories (MaxRkNNT) as follows.

Definition 1 (MaxRkNNT). *Let $\mathcal{D}$ be a trajectory dataset, where $\mathcal{D}_\mathcal{U}$ is a set of user trajectories, and $\mathcal{D}_\mathcal{F}$ is a set of existing facility trajectories. $\mathcal{Q}_\mathcal{F}$ is a set of query trajectories on a shared data space. Each trajectory $\mathcal{T} \in \mathcal{D}$ and $\mathcal{Q}_\mathcal{F}$ is a sequence of locations $\mathcal{T} = \{t_1, t_2, ..., t_n\}$; $n \geq 2$. Each $\mathcal{U} \in \mathcal{D}_\mathcal{U}$ represents a user's travel route and each $\mathcal{F} \in \mathcal{D}_\mathcal{F}$ represents stoppage locations of a facility trajectory. The RkNNT($\mathcal{T}$) finds a subset of $\mathcal{D}_\mathcal{U}$ that take $\mathcal{T}$ as one of their top-k (k nearest) trajectories. A MaxRkNNT query finds a $\mathcal{Q} \in \mathcal{Q}_\mathcal{F}$ for which $|RkNNT(\mathcal{Q})| \geq |RkNNT(\mathcal{Q}')|$, $\forall\ \mathcal{Q}' \in \mathcal{Q}_\mathcal{F} \setminus \mathcal{Q}$.*

In the context of **Fig. 1** MaxRkNNT would output $\mathcal{Q}_2$ since it has the maximal RkNNT set cardinality. To find the top-k facility trajectories for a user trajectory, we need to first define a scoring function (or a distance function) that gives a ranking score between a user trajectory and facility trajectory. To define the distance function $d_t(\mathcal{U}, \mathcal{F})$ between a user trajectory $\mathcal{U}$ and a facility trajectory $\mathcal{F}$, we consider three separate factors that affects the scoring: (i) the distance between $\mathcal{U}$ and the nearest pick-up (or drop-off) point of $\mathcal{F}$, (ii) the travel length of user trajectory $\mathcal{U}$ through facility trajectory $\mathcal{F}$ (iii) user's preference (or weights w) for each point $u_i \in \mathcal{U}$.

Definition 2 (Distance Function). *Let $\mathcal{U}_s$ and $\mathcal{U}_e$ be the start and end locations of a user trajectory $\mathcal{U}$. Function $\eta(p, \mathcal{F})$ returns the nearest pick-up or drop-off point of facility trajectory $\mathcal{F}$ from a location point p of user trajectory $\mathcal{U}$, $l_t(\mathcal{U}, \mathcal{F})$ returns the travel length of a user through trajectory $\mathcal{F}$, and $d_e(a, b)$ is the euclidean distance between any two location points a and b.*

(i) When a user is only interested about her distance from source (destination) to pick-up (drop-off) point, then we can define the distance between a user trajectory $\mathcal{U}$ and facility trajectory $\mathcal{F}$ as follows.

$$d_t(\mathcal{U}, \mathcal{F}) = d_e(\mathcal{U}_s, \eta(\mathcal{U}_s, \mathcal{F})) + d_e(\mathcal{U}_e, \eta(\mathcal{U}_e, \mathcal{F})) \quad (1)$$

(ii) A user may also consider the travel length with the facility. For such as a case, we define the distance function as a combination of travel distance with the facility and the travel distance from source (destination) to pick-up (drop-off) point:

$$d_t(\mathcal{U}, \mathcal{F}) = d_e(\mathcal{U}_s, \eta(\mathcal{U}_s, \mathcal{F})) + d_e(\mathcal{U}_e, \eta(\mathcal{U}_e, \mathcal{F})) + l_t(\mathcal{U}, \mathcal{F})) \quad (2)$$

(iii) A user may have different priorities for different locations on her travel route. Let us assume, each location $u_i \in \mathcal{U}$ has an associated $u_i.w$. Then we can define the distance function for this case as follows.

$$d_t(\mathcal{U}, \mathcal{F}) = \sum_{i=1}^{n} d_e(u_i, \eta(u_i, \mathcal{F})) * u_i.w \quad (3)$$

Since trajectories $\mathcal{U}$ and $\mathcal{F}$ may not have the same number points, we normalize it in the above equation .

Table 1: Notation

Symbol	Description
$\mathcal{D}_\mathcal{U}$	The set of user trajectories
$\mathcal{D}_\mathcal{F}$	The set of facility trajectories
$\mathcal{Q}_\mathcal{F}$	The set of query trajectories
$d_t(\mathcal{U}, \mathcal{F})$	Distance of a facility trajectory from user trajectory
$l_t(\mathcal{U}, \mathcal{F})$	Travel length of user through facility $\mathcal{F}$
$d_e(p, q)$	Eulclidean distance between two points p and q
$d_{min}(p, \mathcal{N})$	Min distance of Quad-tree node $\mathcal{N}$ from point p
$min_e(u, \mathcal{S})$	Min euclidean distance between u_i and set of points $\mathcal{S}$
$\eta(p, \mathcal{F})$	nearest point of $\mathcal{F}$ from a location point $p \in \mathcal{U}$

4 Processing of Top-k (k-NN)

In this section we describe methodologies to determine the top-k nearest facilities of a user trajectory. We first adopt the best-first approach from Tang et al. [11] to incrementally find the next nearest facility w.r.t. a user trajectory. We index all facility points of facility trajectory dataset using a hierarchical index (e.g., Quad-tree/R-tree). Then we incrementally retrieve nearest facility trajectories from a given user trajectory until we find the top-k trajectories for the user. A major challenge in this process is to derive an early termination strategy that avoids the computation of similarities between the user and a large number of trajectories. We propose a lower bound distance, *lbd*, based on the knowledge of the already explored data space and propose an early termination of search based on the computed *lbd*, which reduces the computation overhead significantly.

Lower Bound Distance. The lower bound distance, *lbd*, can be expressed as follows: Let $\mathcal{U} = \{u_1, u_2, \cdots, u_{|\mathcal{U}|}\}$ be a user trajectory. For each u_i, let $fmax_i$ be the farthest retrieved facility point. Thus an unknown facility trajectory will have at least $\sum_i^{|\mathcal{U}|} d_e(u_i, fmax_i)$ distance from $\mathcal{U}$. Then, $\sum_i^{|\mathcal{U}|} d_e(u_i, fmax_i)$ will be the lower bound distance, *lbd*. Based on this lower bound distance, we can describe the top-k processing steps as follows.

First, for a given user trajectory $\mathcal{U}$, we incrementally fetch the nearest facility points with respect to every point $u_i \in \mathcal{U}$. This facility point set $\mathcal{FP}$ contains $|\mathcal{U}|$ number of facility points each representing the NN for each $u_i \in \mathcal{U}$. Now, from the fetched facility point set $\mathcal{FP}$, we retrieve all the facility trajectories $\mathcal{FS}'$ that contain a facility point $f \in \mathcal{FP}$. Let $|\mathcal{FS}'| = n'$ be the number of facilities retrieved in the first phase. We compute the distance, between $\mathcal{U}$ and each facility $\mathcal{F} \in \mathcal{FS}'$. We continue the above process by retrieving more facility points, i.e., 2nd NN, 3rd NN, until we find k distinct facility trajectories. Then, we update the best known distance, λ_k to the kth facility trajectories in terms of distance rank according to our distance function. At this stage, we compute the lower bound distance, *lbd* as described above. If the *lbd* is greater than the distance, λ_k, to kth facility trajectory the search terminates. Otherwise, we repeat the above steps of retrieving more facilities w.r.t. user trajectories.

Delayed Retrieval. A major drawback of the above approach is that the algorithm processes a facility trajectory immediately after fetching any of its facility point w.r.t. any point location $u_i \in \mathcal{U}$. As a result, it may process many unnecessary facility trajectories which cannot be part of the result. Thus, we propose a delayed retrieval approach. In this approach, we process a facility if at least τ location points are fetched during the search process.

Similar to the above approach, we search for next nearest facility location point f_i from each user location point $u_i \in \mathcal{U}$, and add the corresponding facility in a candidate set $\mathcal{C}$. Now, candidates in $\mathcal{C}$ can be divided into two sets. First set $\mathcal{C}_\tau$ contains the candidates for which at least τ points have been found during the process and second set $\mathcal{C}_\tau{}'$ contains the candidates of $\mathcal{C} \setminus \mathcal{C}_\tau$. We process the candidate $c \in \mathcal{C}_\tau$ by computing corresponding distance $d_t(\mathcal{U}, c)$. We also update the top-k result list $\mathcal{R}$ and λ_k accordingly. On the other hand, we compute *expected distance* for the candidates in $\mathcal{C}_\tau{}'$, for which less than τ process have been found. Algorithm terminates when λ_k is less than *lbd* or the minimum *expected distance* of the potential candidates in $\mathcal{C}_\tau{}'$.

Expected Distance. Suppose that at some point during the search process for a user trajectory $\mathcal{U} = \{u_1, u_2, \cdots, u_n\}$, $\mathcal{F}_c$ is a potential candidate for which less than τ points have been retrieved (i.e., $\mathcal{F}_c \in \mathcal{C}_\tau{}'$). For each u_i, if $p_i \in \mathcal{F}_c$ be the nearest retrieved point, then $d_e(u_i, p_i)$ contributes to the *expected distance*. Otherwise, $d_e(u_i, fmax_i)$ contributes to the *expected distance* value, where $fmax_i$ is the farthest retrieved point from u_i. Finally, we normalize the distance. We notice that expected distance value must be less than the actual distance between $\mathcal{U}$ and $\mathcal{F}_c$. **Fig. 1** illustrates the computation of Expected Distance for user trajectory $\mathcal{U}_2$ and potential facility candidate $\mathcal{F}_2$ during the top-k facility search for $\mathcal{U}_2$, where each circle represents the distance $d_e(u_i, fmax_i)$.

5 Computing RkNNT

In this section, we describe our algorithm to find the RkNNT set for a query trajectory based on the precomputed k-NN (or top-k) facilities for each user trajectory.

5.1 Algorithm

For a given set of query trajectories $\mathcal{Q}_{\mathcal{F}}$, a straightforward approach requires to compute the distance $d_t(\mathcal{U}, \mathcal{Q})$ of each $\mathcal{Q} \in \mathcal{Q}_{\mathcal{F}}$ from each user trajectory $\mathcal{U} \in \mathcal{D}_{\mathcal{U}}$. If the distance $d_t(\mathcal{U}, \mathcal{Q})$ is less than the distance of k^{th} facility of user $\mathcal{U}$, $\mathcal{U}$ is included as one of the RkNNT of $\mathcal{Q}$.

Baseline Approach. Let Λ_k be the maximum value among the distances of k^{th} facility (λ_k) for all user trajectory $\mathcal{U} \in \mathcal{D}_{\mathcal{U}}$. Also assume that MBR($\mathcal{U}$) and MBR($\mathcal{Q}$) are *minimum bounding rectangles* containing user trajectory $\mathcal{U}$ and query trajectory $\mathcal{Q} \in \mathcal{Q}_{\mathcal{F}}$, respectively. Now for each $\mathcal{Q}$, we search for the user trajectories $\mathcal{U} \in \mathcal{D}_{\mathcal{U}}$ for which the minimum distance between MBR($\mathcal{U}$) and MBR($\mathcal{Q}$) is less than Λ_k. If $d_t(\mathcal{U}, \mathcal{Q})$ is less than $\mathcal{U}.\lambda_k$, we include $\mathcal{U}$ as one of the results of RkNNT for $\mathcal{Q}$.

An R*-Tree Based Approach. As a part of the pre-processing step, we maintain the distance of k^{th} facility (λ_k) of each user trajectory $\mathcal{U}$. Now, to find the result of the RkNNT for a given set of query trajectories $\mathcal{Q}_{\mathcal{F}}$, we construct MBRs containing the points of each $\mathcal{U}$ and we denote the MBR as UMBR. However, a query cannot be considered as a k-NN of a user trajectory if it doesn't contain any point within the λ_k distance of the corresponding user trajectory points. Hence, we extend each UMBR by $\mathcal{U}.\lambda_k$ on each side and denote the extended MBR as E-UMBR. Finally, we build an R*-Tree using the E-UMBR for each user trajectory.

For each query trajectory $\mathcal{Q} \in \mathcal{Q}_{\mathcal{F}}$, we construct MBRs (QMBR) using the points of each $\mathcal{Q}$. If E-UMBR of a user $\mathcal{U}$ does not intersect with the QMBR of a query trajectory $\mathcal{Q}$, then no point of $\mathcal{Q}$ is located within the λ_k distance of the points of user trajectory $\mathcal{U}$. Thus, we can exclude $\mathcal{U}$ from the RkNNT candidate set of $\mathcal{Q}$. On the other hand, If QMBR of a query trajectory $\mathcal{Q}$ intersects E-UMBR of user trajectory $\mathcal{U}$, we consider $\mathcal{U}$ as a potential candidate of RkNNT of $\mathcal{Q}$. If $d_t(\mathcal{U}, \mathcal{Q})$ is less than $\mathcal{U}.\lambda_k$, we include $\mathcal{U}$ as one of the results of RkNNT for $\mathcal{Q}$. Finally we output the $\mathcal{Q}$ with maximum cardinality of RkNNT set to answer MaxRkNNT query. **Fig. 2** illustrates the RkNNT candidate selection strategy for a query trajectory $\mathcal{Q}_1$. QMBR of $\mathcal{Q}_1$ does not intersect the E-UMBR for user trajectory $\mathcal{U}_2$, which excludes $\mathcal{U}_2$ from the candidate set for RkNNT of $\mathcal{Q}_1$. On the other hand, QMBR of $\mathcal{Q}_1$ intersects E-UMBR of $\mathcal{U}_1$ and thus, included as a candidate of for RkNNT of $\mathcal{Q}_1$.

5.2 Updates

Candidate trajectory selection depends on precomputed data i.e. on top-k processing of each user trajectory. A major drawback of using precomputed data is

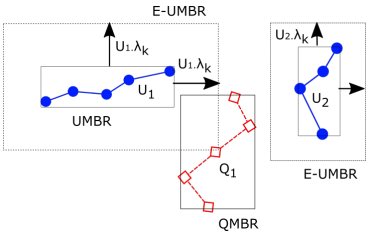


Fig. 2: Candidate trajectory selection using MBR of top-k results of users

that any update in dataset may invalidate the data and trigger a complete re-computation. However, the algorithms presented in this paper are designed such that any update in dataset are handled without repeating the precomputation step. We may wish to add new user and/or facility trajectories to the existing dataset and run MaxRkNNT query. We may also want to change the value of k at some point. While we make these changes we do not want to recompute for the whole dataset, allowing a user the flexibility to tune parameters and get the query results in run-time. In this section we discuss how the algorithm handles these updates.

Update k. If k is increased to k', $k' - k$ facility trajectories must be added to top-k list of each user trajectory. Since at this point we have λ_k of each $\mathcal{U}$ and any further facility must be at a distance $\geq \lambda_k$, we only need to consider next nearest facility locations that is at least λ_k distance away from a user point. While traversing the quad tree for next nearest neighbor, nodes at distance $< \lambda_k$ can be safely pruned. No computation is required if k is decreased.

Add/Remove Facility Trajectory. Addition of facility trajectory $\mathcal{F}$ to $\mathcal{D}_{\mathcal{F}}$ may update top-k list of one or more user trajectories $\mathcal{U}$. We use R*-Tree Based algorithm RkNNT query to efficiently find the set of user trajectories that now takes $\mathcal{F}$ as one of their top-k nearest facilities. If some $\mathcal{F}$ is removed from dataset, we search RkNNT of $\mathcal{F}$. It is then removed from top-k facilities of these user trajectories. This creates an empty slot in top-k and a next nearest facility trajectory is added using the process of updating k described above.

Add/Remove User Trajectory. Adding a user trajectory $\mathcal{U}$ to $\mathcal{D}_{\mathcal{U}}$ requires finding top-k nearest facilities. We use Delayed Retrieval based top-k search algorithm for this since top-k search for $\mathcal{U}$ is independent of searches for $\mathcal{U}' \in \mathcal{D}_{\mathcal{U}} \setminus \mathcal{U}$. Removing $\mathcal{U}$ would not have any effect on the precomputed data of other user trajectories.

6 Experimental Evaluation

In this section, we describe the experimental evaluation of the algorithms for processing the query with both real and synthetic datasets and compare them with the baseline approach.

User Trajectory Dataset ($\mathcal{D}_{\mathcal{U}}$). We use T-Drive dataset[2] for real user trajectories. The dataset contains one-week trajectories of $10,357$ taxis from Beijing. Total number of points in the dataset is about 15 million and total distance covered by the trajectories is 9 million kilometers. For our experiments, we do periodic sampling for larger trajectories so that no user trajectory contains more than 10 points. Thus, our final dataset contains $165,736$ user trajectories.

Facility Trajectory Dataset ($\mathcal{D}_{\mathcal{F}}$). We download the OpenStreetMap[3] data within the same *latitude* and *longitude* range (Beijing city) of T-drive dataset. Then, we parse the location points of the nodes under the $<way>$ tags to construct the facility trajectories. We ignore the tags that contain less than 10 points. After final processing, our dataset contains $27,876$ facility trajectories.

Query Trajectory Dataset ($\mathcal{Q}_{\mathcal{F}}$). We prepared our query dataset by randomly generating trajectory locations within the same region as users and facilities. We randomly chose points located within the region and appended new points one by one with a limited rotation angle and step size.

Synthetic Dataset. We also evaluate our algorithm on synthetic dataset where trajectories are generated randomly within a specified area. We randomly select the start point for each trajectory and append new points one by one, while selecting the rotation angle randomly within $90°$ to avoid zigzag trajectories.

All experiments were performed on an Intel Core i5-6200U CPU with 8GB RAM running Ubuntu 16.04 LTS, implemented in python 2.7. Table 2 shows the parameter setting used in the experimental evaluation.

Parameter	Range	Default
Number of Query, ($\lvert\mathcal{Q}_{\mathcal{F}}\rvert$)	10, 20, 30, 40, 50, 60, 70	20
Number of Facility Point	10, 20, 30, 40, 50, 60	30
Retrieval Threshold, (τ)	1, 3, 5, 7, 9, 11, 13, 15, 17	9
k	5, 10, 15, 20, 25	5

Table 2: Parameter Setting

6.1 Top-k Processing

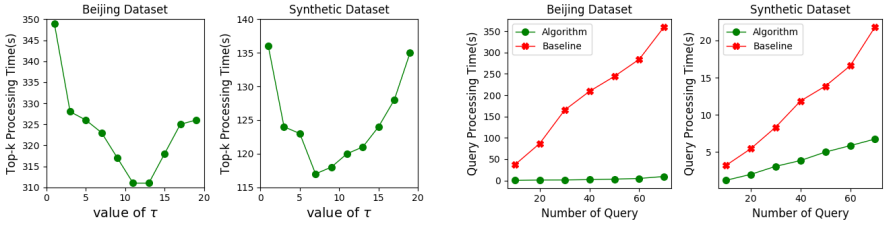
We compare delayed retrieval approach for top-k processing with baseline approach since the brute-force approach is not scalable. Delayed retrieval clearly outperforms the baseline approach by 25-30% as expected since it reduces unnecessary trajectory distance computations. Table 3 shows the time in seconds (s) for both approaches for default values of parameters. It takes 5.1 hours to perform Top-k processing for the entire T-drive dataset and 6.5 hours for synthetic

[2] https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/

[3] http://www.openstreetmap.org

$\mathcal{D}_\mathcal{U}$	**Delayed Retrieval**	**Baseline**	$\mathcal{D}_\mathcal{F}$	**Delayed Retrieval**	**Baseline**
20,000	3,043s	4,126s	500	1,365s	1,784s
40,000	7,801s	9,872s	1,000	1,820s	2,324s
60,000	16,203s	21,147s	1,500	2,231s	3,031s
k= 5, $\mathcal{D}_\mathcal{F}$ = 5,500			k= 5, $\mathcal{D}_\mathcal{U}$ = 20,000		

Table 3: Top-k Processing Time in seconds (s) for both Delayed Retrieval and Baseline approach using Beijing Dataset.



(a) (b)

Fig. 3: (a) Effect on top-k processing time with the increase of retrieval threshold τ (b) Effect on MaxRkNNT query processing time with the increasing number of queries $|\mathcal{Q}_\mathcal{F}|$

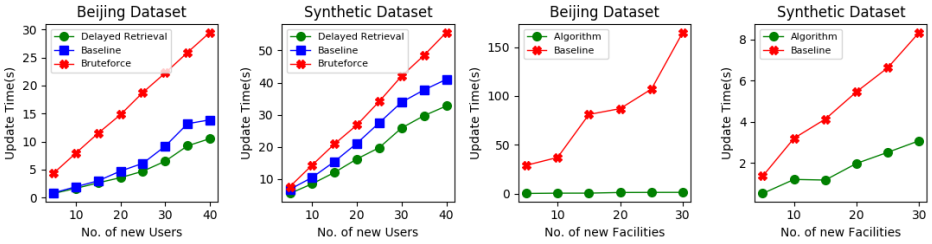


Fig. 4: Update time with the increasing number of new User and Facility Trajectories for Beijing dataset and Synthetic dataset.

dataset. We vary several parameters for better insights of the top-k processing approaches. For the following experiments of top-k processing we use a subset of $\mathcal{D}_\mathcal{U}$ and $\mathcal{D}_\mathcal{F}$ to focus more on the graph trend than dataset size.

Varying τ. Fig. 3a shows the effect of varying τ, the threshold amount of points for delayed retrieval in top-k processing time. For both datasets, the runtime starts to decrease with increasing value of τ and starts increasing after a certain point. This is because with a small τ, distance to a facility trajectory $\mathcal{F}$ is computed as soon as a small amount of points of $\mathcal{F}$ are found. This results in a large number of distance computation for trajectories that finally may not be in top-k. For a large τ, the algorithm has to delay computation until a large number of points of $\mathcal{F}$ are found.

Varying No. of Facility Trajectory Points. We compute the top-k processing time by changing the number of facility points from 10 points to 60 points. To alter the facility point, we drop extra points while reading the trajectory from dataset file. Even though increasing the number of facility points yields increased amount of time during trajectory distance computation, it does not have significant effect on overall top-k processing time, as our algorithm reduces the number of unnecessary distance computation. For each dataset, delayed retrieval approach gives 25-30% improvement over the computation time for baseline approach.

6.2 MaxRkNNT Processing

We evaluate our proposed algorithm to answer MaxRkNNT, and compare the performance with baseline approach.

Varying Number of Queries $|\mathcal{Q}_{\mathcal{F}}|$. Fig. 3b illustrates the effect of changing query size $|\mathcal{Q}_{\mathcal{F}}|$ in the query answering time. For both datasets R*-Tree approach outperforms the baseline. With the change of $|\mathcal{Q}_{\mathcal{F}}|$, our algorithm causes very small-scaled changes in query time. Baseline approach on the other hand, requires significant amount of CPU time to answer respective queries. Since our algorithm significantly reduces the search space for the queries, it takes very small CPU time in answering even a larger query set.

6.3 Update Processing

Updating Trajectories. Fig. 4 illustrates the computation time required for updating new user and facility trajectories. To update new user trajectories, we only compute top-k for each new trajectory. Since baseline approach requires computing larger number of trajectory distance than Delayed Retrieval approach, it takes increased amount of CPU time to update new user trajectory. On the other hand, brute-force approach computes distance from the new user trajectories to all existing facilities, it gives poor performance during update. When new facility arrives, we need to find whether those facilities belong to the top-k facilities of any of the existing users. We use RkNNT query to find the users who has the new facility trajectories as one of their k-nearest neighbors trajectories. Finally, we update the top-k facilities of the corresponding users.

Updating k. When the value of k is increased, we avoid recomputing top-k facilities for each user by storing the state of the previous search for old value of k. We restore the priority queues of each users to resume the search to find the further top-k facilities.

7 Conclusion

In this paper, we have proposed efficient solution for Maximizing Reverse k-Nearest Neighbors for Trajectories (**MaxRkNNT**) query. We first introduced a generic similarity measure between a multi-point query trajectory and a multi-point facility trajectory that helps us to define the nearest neighbors according to

user requirements. Then, we have proposed pruning strategies that can quickly compute k-NNs (or top-k) facility trajectories for a given user trajectory. Finally, we have developed a filter and refinement technique to compute the MaxRkNNT. Our experimental results show that our proposed approach significantly outperforms the baseline for both real and synthetic datasets.

References

1. Lei Chen and Raymond T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.
2. Lei Chen, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *ACM SIGMOD*, pages 491–502, 2005.
3. Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, Yu Zheng, and Xing Xie. Searching trajectories by locations: an efficiency study. In *ACM SIGMOD*, pages 255–266, 2010.
4. King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record*, 27(3):16–21, 1998.
5. Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *GeoInformatica*, 11(2):159–193, 2007.
6. Yubao Liu, Raymond Chi-Wing Wong, Ke Wang, Zhijie Li, Cheng Chen, and Zitong Chen. A new approach for maximizing bichromatic reverse nearest neighbor search. *Knowl. Inf. Syst.*, 36(1):23–58, 2013.
7. Radi Muhammad Reza, Mohammed Eunus Ali, and Muhammad Aamir Cheema. The optimal route and stops for a group of users in a road network. In *SIGSPATIAL/GIS*, pages 4:1–4:10. ACM, 2017.
8. Samia Shafique and Mohammed Eunus Ali. Recommending most popular travel path within a region of interest from historical trajectory data. In *MobiGIS*, pages 2–11. ACM, 2016.
9. Shuo Shang, Bo Yuan, Ke Deng, Kexin Xie, and Xiaofang Zhou. Finding the most accessible locations: reverse path nearest neighbor query in road networks. In *GIS*, pages 181–190. ACM, 2011.
10. Zhexuan Song and Nick Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.
11. Lu An Tang, Yu Zheng, Xing Xie, Jing Yuan, Xiao Yu, and Jiawei Han. Retrieving k-nearest neighboring trajectories by a set of point locations. In *SSTD*, pages 223–241, 2011.
12. Michail Vlachos, Dimitrios Gunopulos, and George Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.
13. Sheng Wang, Zhifeng Bao, J. Shane Culpepper, Timos K. Sellis, and Gao Cong. Reverse k nearest neighbor search over trajectories. *CoRR*, abs/1704.03978, 2017.
14. Raymond Chi-Wing Wong, M. Tamer Özsu, Philip S. Yu, Ada Wai-Chee Fu, and Lian Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.
15. Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.
16. Zenan Zhou, Wei Wu, Xiaohui Li, Mong-Li Lee, and Wynne Hsu. Maxfirst for maxbrknn. In *ICDE*, pages 828–839. IEEE Computer Society, 2011.