The Maximum Trajectory Coverage Query in Spatial Databases

Mohammed Eunus Ali
BUET, Bangladesh
eunus@cse.buet.ac.bd

Shadman Saqib Eusuf
BUET, Bangladesh
s.saqibeusuf@gmail.com

Kaysar Abdullah
BUET, Bangladesh
kzr.buet08@gmail.com

Farhana M. Choudhury
RMIT University and University of Melbourne, Australia
fchoudhury@unimelb.edu.au

J. Shane Culpepper
RMIT University, Australia
shane.culpepper@rmit.edu.au

Timos Sellis
Swinburne University of Technology, Australia
tsellis@swin.edu.au

ABSTRACT

With the widespread use of GPS-enabled mobile devices, an unprecedented amount of trajectory data has become available from various sources such as Bikely, GPS-wayPoints, and Uber. The rise of smart transportation services and recent break-throughs in autonomous vehicles increase our reliance on trajectory data in a wide variety of applications. Supporting these services in emerging platforms requires more efficient query processing in trajectory databases. In this paper, we propose two new coverage queries for trajectory databases: (i) k *Best Facility Trajectory Search* (kBFT); and (ii) k *Best Coverage Facility Trajectory Search* (kBCovFT). We propose a novel index structure, the Trajectory Quadtree (TQ-tree) that utilizes a quadtree to hierarchically organize trajectories into different nodes, and then applies a z-ordering to further organize the trajectories by spatial locality inside each node. This structure is highly effective in pruning the trajectory search space, which is of independent interest. By exploiting the TQ-tree, we develop a divide-and-conquer approach to efficiently process a kBFT query. To solve the kBCovFT, which is a *non-submodular NP-hard problem*, we propose a greedy approximation. We evaluate our algorithms through an extensive experimental study on several real datasets, and demonstrate that our algorithms outperform baselines by two to three orders of magnitude.

PVLDB Reference Format:
Mohammed Eunus Ali, Shadman Saqib Eusuf, Kaysar Abdullah, Farhana M. Choudhury, J. Shane Culpepper, and Timos Sellis. The Maximum Trajectory Coverage Query in Spatial Databases. *PVLDB*, 12(3): 197-209, 2018.
DOI: https://doi.org/10.14778/3291264.3291266

1. INTRODUCTION

With the widespread use of GPS-equipped mobile devices and the popularity of mapping services, an unprecedented amount of trajectory data is becoming available for data analytics applications. For example, in Bikely[1] users can share their cycling routes from the GPS devices, in GPS-wayPoints[2] a user can add waypoints (points on a route at which a course is changed) in a route and share with friends, in Microsoft GeoLife[3] users can share their travel routes and experience using GPS trajectories. Most of the popular social network sites also support sharing user trajectories.

Figure 1: An example of a kBFT query and a kBCovFT query with 12 user trajectories and 3 bus routes in NY, USA

Beyond personal trips and travel routes, there are many examples of trajectories derived from various transport services. Uber served nearly 14.3 million users in New York City between January-June 2015[4]. While user trajectory data has already been used for public transport planning, a wide range of applications remain where planning ad-hoc transport services are of interest. As discussed in an IEEE Spectrum report earlier this year, ride-sharing, taxi services, and on-demand transportation services will be key sectors in the emerging autonomous vehicle industry [18]. Consider the following examples that highlight potential applications in ad-hoc transportation service planning.

Scenario 1: An on-demand bus operator wants to run autonomous vehicles on different routes of a city based on commuter demand. Commuters can request transport services by submitting their pick-up and drop-off locations (and their expected travel times). Assume that the operator has a limited number (k) of vehicles operating on n predefined routes on public transportation networks in a city.

[1] http://www.bikely.com
[2] http://gpswaypoints.net
[3] https://research.microsoft.com/en-us/projects/geolife/
[4] https://github.com/fivethirtyeight/uber-tlc-foil-response

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291266

Now at any time, the bus operator may need to know the top-k routes that serve the maximum number of users.

Scenario 2: Consider a tourist city, where each tourist has a list of POIs to visit. Now, a tour operator wants to run buses for k different routes to serve the maximum number of tourists. Tourists would use the service if the number of POIs that can be visited from their list is maximized.

The underlying problem in both the above scenarios is to select a limited number (top-k) of facility trajectories/routes from a given set that best "serve" user trajectories. In Scenario 1, only the start and end points of each user trajectory are of interest, and a user is serviced (e.g., will ride the bus) if a bus stop is within a certain distance ψ of the desired locations. In this case, service of a facility is a binary notion; a user is either served or not. In addition to distances to the nearby stops, a user may also use the facility if the travel route (or travel time) does not deviate much from the original route. In Scenario 2, all the points in a user trajectory can be important as one may want to maximize the "service" by a facility trajectory in terms of the number of points (e.g., the number of POIs that a tourist can visit) or the trajectory length (e.g., the length of a journey). In this case, a user can be served partially by a facility. We use the term "service" to refer to both these binary and non-binary measures (details in Section 2).

In this paper, we address this new class of trajectory search problems, denoted as a k **Best Facility Trajectory Search (kBFT)** query which finds k facilities that maximize a service measure for a set of user trajectories. Formally, given a set U of user trajectories, a set F of candidate facility trajectories, and a positive integer k, a kBFT query returns k facilities from F with the highest service to U. We also address another variant of the query, the k **Best Coverage Facility Trajectory Search (kBCovFT)** that returns k facilitates from F that *combinedly* serve the maximum number of user trajectories from U. As the service value, which conceptually is how well users are served by a facility, may vary across applications, we formally define a service value function $SO(U, f)$ to measure the service of a facility f on user trajectories U. For the kBCovFT problem, the service value $SO(U, F')$ is computed for a subset of facilities $F' \subseteq F$, where the common service provided by different facilities to the users are considered (Section 2 for details).

EXAMPLE 1. *Figure 1 shows an example of a kBFT query for user trajectories* $\{u_1, \ldots, u_{12}\}$ *and facility trajectories* $\{25, 46, 65\}$ *representing the bus routes with stop points in Queens, NY. A user will use a facility if there is a pickup/drop-off location of that facility within a threshold distance from her source and destination. Thus,* u_1, u_2, u_4 *can be served by* 25*,* $u_5, \ldots, u_8$ *by* 46*, and* u_9, u_{12} *by* 65*. Hence the bus route* 46 *is the result for k=1. For a kBCovFT query, since* u_{10}, u_{11} *can be served jointly by* 46 *and* 65*, for* $k = 2$ *the answer is* $\{46, 65\}$ *as they can serve* $\{u_5, \ldots, u_{12}\}$*, where the other sets of size* 2*,* $\{25, 46\}$ *and* $\{25, 65\}$ *can serve* $\{u_1, u_2, u_4, \ldots, u_8\}$*, and* $\{u_1, u_2, u_4, u_9, u_{12}\}$*, respectively.*

A major challenge with these queries is to quickly find co-located and similarly oriented user trajectories that can be served by a facility. Moreover a user can be served partially by a facility, and a user can be served by multiple facilities; hence another major challenge is to track the different segments of a trajectory that can be served by a facility trajectory. In most of the existing work on trajectories ([29, 16]), the points of the trajectories are indexed using a state-of-the-art spatial indexing method. However, such techniques are not amenable to our problem as both the partial service of a user trajectory (e.g., the number of POIs from the list of interesting places that a tourist can visit), and the combined service of multiple facilities are required in this problem. Similarly, previous studies that find trajectories within a range of a query trajectory ([28]), or find the reverse k nearest neighbor trajectories ([32]) cannot be used, as it would require inefficiently repeating the approaches for each facility route. Moreover, to the best of our knowledge, there is no existing work on trajectories that can be used to efficiently answer kBCovFT, where a user trajectory can be served jointly by multiple facility trajectories (Section 8 for details).

To alleviate the above limitations, we first propose a novel two-level index structure, the Trajectory Quadtree (TQ-tree) that facilitates efficient processing of kBFT and kBCovFT queries. The novelty of our work comes from the key observation that if the points of multiple user trajectories are co-located and have a similar orientation, then those trajectories are likely to be served by the same facility. Hence, we store co-located, similar-extent, and similar-oriented trajectories together in a TQ-tree node. Specifically, a quadtree structure is employed to organize the trajectories in a hierarchy based on their extent and orientation, and then a z-ordering is applied to organize the trajectories by spatial locality inside a quadtree node. Such a structure is highly effective in pruning the search space for different segments of trajectories based on locality and orientation, which is of independent interest. Note that although the TQ-tree is primarily designed for processing facility trajectory queries, this index is also suitable for processing other important queries such as trajectory similarity join queries [26].

Next, we present an efficient divide-and-conquer approach to answer kBFT queries, where a facility trajectory is recursively divided and the service value of the components of the facility is calculated for that subspace. For each subspace, we apply a two-phase pruning technique using the TQ-tree. As either the partial or the complete service values are important based on the application, we present a best-first strategy to efficiently process the facilities using the service upper bound value. We also present different conditions where the process can be early terminated safely. By exploiting the pruning formulation of the kBFT query solution, we provide an efficient approximation algorithm for processing kBCovFT, which we prove to be a non-submodular NP-hard problem. The contributions of the paper are summarized as follows:

- We propose a new class of trajectory queries: (i) kBFT and (ii) kBCovFT.
- We propose a novel two-level index structure, the Trajectory Quadtree (TQ-tree) based on the co-location and similar-orientation characteristics of user trajectories.
- We present an efficient divide-and-conquer approach for answering kBFT queries using the TQ-tree, which deploys a two-phase pruning technique.
- We prove that kBCovFT is a non-submodular NP-hard problem. We propose an efficient two-step greedy approximation algorithm to answer kBCovFT.
- We show that our approaches work for trajectories in both Euclidean and road network data spaces, and can handle temporal dimension in trajectories.

2. PROBLEM FORMULATION

Let U be a set of user trajectories where each $u \in U$ is a sequence of point locations, $u = \{p_1, p_2, ..., p_{|u|}\}$ and F be a set of facility trajectories, where each $f \in F$ is a sequence of stop points representing the pick-up or drop-off locations of a facility route (e.g., bus route). A user trajectory can be served by a facility in different contexts. First, we present the calculation of the service values of a facility for a single user in different scenarios, and then we present a generalized function to compute the service value of a facility or a set of facilities for the set of users U.

2.1 Service value for a single user

Binary Service Function. Here, $u.p_1$ and $u.p_{|u|}$ are the source and destination locations of u. A user u may only be interested in using a facility f if there is any stop point of f within a certain distance from the source and destination of u, i.e., $dist(u.p_1, f) \leq \psi \wedge dist(u.p_{|u|}, f) \leq \psi$. Alternatively, a user u might be only interested in using a facility f if the total distance from the source and destination of u to the nearest the nearest stop points of f is within a certain distance ψ, where $dist(u.p_1, f) + dist(u.p_{|u|}, f) \leq \psi$. Here, ψ can be set based on the distance/range that a user can cover on foot or by other means for availing the transportation facility. In such cases, we can define the Boolean service function $S(u, f)$ as:

$$S(u, f) = \begin{cases} 1 & \text{if } u \text{ is served by } f \\ 0 & \text{otherwise.} \end{cases}$$

Note that, in the above scenarios, we consider a binary notion of service, where a user is either served or not-served. However, in some applications, one may want to evaluate a service score based on how well a facility serves a user. In that case, we can compute a service score between 0 and 1 based on the proximity of the user from the stop points. In addition to the distance limit to the stop points, a user might be only interested in using a facility if the travel length using the facility does not deviate much from the current travel length using their car.

Non-binary Service Functions. In non-binary cases where u can be served partially by f, the service can be computed based on the number of points in u that can be served by f, $scount(u, f)$ (e.g., the number of POIs that can be visited by a tourist). Then the service value of f is calculated as: $S(u, f) = \frac{scount(u, f)}{|u|}$.

Formally, $scount(u, f) = \sum_{p_i \in u} S_c(u.p_i, f)$ where,

$$S_c(u.p_i, f) = \begin{cases} 1 & \text{if } dist(u.p_i, f) \leq \psi \\ 0 & \text{otherwise.} \end{cases}$$

When the interest is in maximizing the length of u served by f, $slength(u, f)$ (e.g., the length of journey with advertisement display), the service value is calculated as: $S(u, f) = \frac{slength(u, f)}{length(u)}$, where $length(u)$ is the total length of u. Note that the length of two trajectories with the same number of points can be different based on the length of the segments between those points.

Formally, $slength(u, f) = \sum_{i=1}^{|u|-1} S_l(u.p_i, u.p_{i+1}, f)$. Here, $S_l(u.p_i, u.p_{i+1}, f) = len(u.p_i, u.p_{i+1})$, when $dist(u.p_i, f) \leq \psi \wedge dist(u.p_{i+1}, f) \leq \psi$; otherwise, $S_l(u.p_i, u.p_{i+1}, f) = 0$.

2.2 Service value for the set of users

As the objective of a facility is to maximize the service to U, the service value of a facility f for U is calculated as:

$$SO(U, f) = \sum_{u \in U} S(u, f) \tag{1}$$

For a collection F' of facilities, where $F' \subseteq F$, we can generalize the service value function as follows:

$$SO(U, F') = \sum_{u \in U} AGG_{f \in F'} S(u, f) \tag{2}$$

Since a user can be served by more than one facility in F', we only consider the service once if the same service is provided by more than one facility. The function *AGG* takes this issue into account by aggregating the services provided by each $f \in F'$ to u.

Although the AGG function is straightforward with a binary service function, it is non-trivial for non-binary services. Let u_1 be a user trajectory with three points p_1, p_2, p_3. Assume that (p_1, p_2) is served by f_1 and (p_2, p_3) is served by f_2. In this case, half of the service value for u_1 is accounted for by f_1, and the remaining by f_2. Now, when we combine the service value of f_1 and f_2 by using our AGG function, we simply add these service values to get the aggregate service value of both facilities. Again, if another facility f_3 serves (p_2, p_3), and if we need to find the aggregate service value of f_1, f_2 and f_3, we need to count this service value for (p_2, p_3) only once as the same portion of the trajectory is served by both f_2 and f_3. In this later case, if we consider non-binary service function, we will count the service value of the facility which yields maximum between f_2 and f_3 while serving (p_2, p_3).

Problem definition. Based on the above definitions, we formally define our trajectory queries as follows.

DEFINITION 1. *(kBFT). Given a set U of user trajectories, a set F of facilities, a positive integer k, and a service value function* $\text{SO}(\cdot)$*, a kBFT query returns the top-k facilities F' from F such that* $\forall f' \in F', \forall f \in F \setminus F', \text{SO}(U, f') \geq \text{SO}(U, f)$.

DEFINITION 2. *(kBCovFT). Given a set U of user trajectories, a set F of facilities, a group size k, and a service value function* $\text{SO}(\cdot)$*, let the set* SG_k *be all possible subgroups of size k from F. The kBCovFT query returns a subgroup* $\text{sg} \in \text{SG}_k$ *of facilities such that for any other subgroup* $\text{sg}' \in \text{SG}_k \setminus \{\text{sg}\}$, $\text{SO}(U, sg) \geq \text{SO}(U, \text{sg}')$.

3. TRAJECTORY QUAD-TREE (TQ-TREE)

The key observation behind our proposed index is, the trajectories whose points are co-located, are likely to use the same facility. Thus such trajectories should be stored together. Based on this observation we present a novel index, the Trajectory Quad (TQ) tree, where trajectories with *close spatial proximity and similar orientation* are grouped and stored together in an effective way. For simplicity, we first describe the index for trajectories with two endpoints (source-destination), and later we generalize for trajectories with any number of points. A two-level indexing is applied to index the trajectories in a TQ-tree. We explain the index construction process and the rationale behind each step in the following.

Hierarchical organization. The space is recursively partitioned to group spatially similar trajectories together. Specifically, a quadtree structure is employed to partition the space. Each node E of the quadtree, denoted as a q-node is associated with a pointer to a list $UL(E)$ of user trajectories. If E is a leaf node, $UL(E)$ contains the intra-node trajectories, which are the trajectories whose both endpoints reside in E. Otherwise, $UL(E)$ consists of the inter-node user trajectories, which are trajectories whose two endpoints reside in two immediate child nodes of E. A node of the quadtree is partitioned until there is no such inter-node trajectories left to be stored with that node, or contains at most β number of intra-node trajectories. Here, β corresponds to the size of a memory block (or a disk block for a disk-resident list $UL(E)$).

With each q-node E, an upper bound, s_{ub} of the service value is stored for the trajectories stored in the subtree rooted at E. For Scenario 1 , s_{ub} of E is the total number of user trajectories, and for Scenario 2, s_{ub} is the total number of points of the user trajectories, in the sub-tree rooted at E, respectively.

As mentioned in prior work [31], one of the major challenges of indexing trajectories is in organizing trajectories of different lengths. Unlike traditional spatial hierarchical indexing, where only the leaf nodes contain the data, we store the trajectories in both leaf

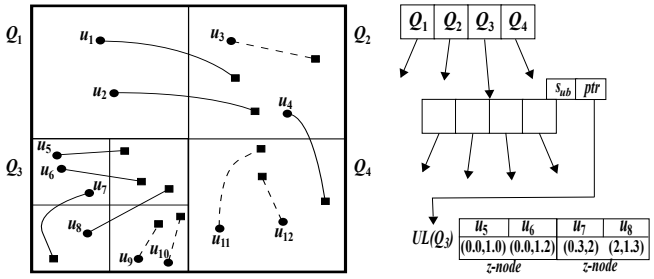


Figure 2: A TQ-tree structure for trajectories.

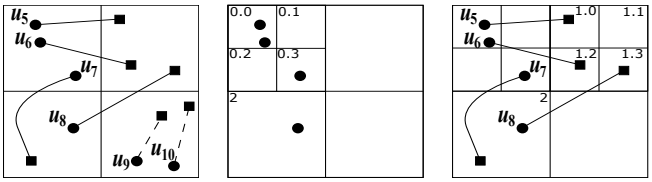


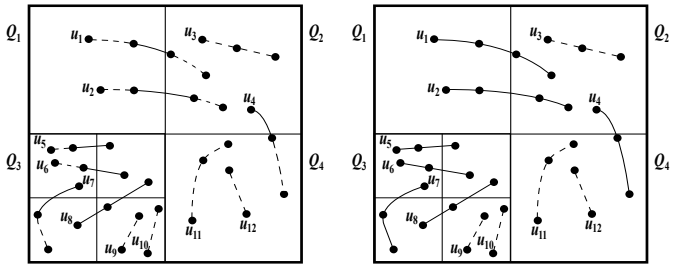
Figure 3: (a) Inter-node trajectories of Q_3 (solid lines), (b) z-ordering of start points, (c) z-ordering of end points.

and non-leaf nodes. In this hierarchical organization, longer trajectories are more likely to be stored in upper level nodes and shorter trajectories in lower level nodes. Such an organization will later facilitate efficient pruning and service (either partial or complete) calculations for both longer and shorter trajectories.

EXAMPLE 2. *Figure 2 shows an example TQ-tree for the user trajectories, $\{u_1, \ldots, u_{12}\}$, where $\beta = 2$. The space is first divided into $Q_1, \ldots, Q_4$. As Q_4 only contains β intra-node trajectories, and the trajectories in Q_1 and Q_2 are stored as the inter-node trajectories of the root node, these q-nodes are not partitioned further. Q_3 is further divided into four quadrants. The inter-node trajectories of Q_3 are $u_5, \ldots, u_8$, and the partitioning terminates.*

Ordered bucketing using z-curve. Depending on the application scenarios and the user travel patterns, the list of trajectories in a q-node can be quite large. For example, if there are many users who travel every day from the same suburb to the city, these user trajectories may all fall under a particular q-node. Thus, storing these trajectories as a flat list may result in poor performance. Therefore we use a space filling curve, specifically a z-curve (Morton order) to order the trajectories such that the trajectories with *close spatial proximity and similar orientation* are grouped together into a single "bucket". The list $UL(E)$ of each q-node is arranged as a sorted list of buckets, where the trajectories in each bucket is also sorted by their z-ordering. Here each bucket is referred to as a *z-node*.

Specifically, for each q-node E: (i) We first apply the z-ordering on the start points of the user trajectories in $UL(E)$. The space enclosed by E is partitioned until each partition contains at most β start points. (ii) Then, we partition the space based on the end points of the user trajectories, where each partition can contain a maximum of β end points. If multiple trajectories have the same z-id for their start points, the space is partitioned until the end point of each such trajectory is assigned a different z-id. This step enables us to distinguish between the trajectories with the co-located start points. (iii) Based on the z-order numbers assigned to each points of user trajectories, we keep them in a sorted bucket list, where each bucket can contain at most β trajectories. If each trajectory is an ordered sequence of points, then we order the trajectories based on the starting point first, and if two trajectories have the same z-orders, we order them based on their second points, and so on. If the trajectories are defined as non-ordered sequence of points, then we order the points of a trajectory based on the z-order, and then apply the aforementioned procedure to sort the trajectories.



(a) Segmented TQ-tree (b) Full-trajectory TQ-tree

Figure 4: Multiple-point Trajectories in a TQ-tree (solid lines are inter-node, and dashed lines are intra-node segments).

EXAMPLE 3. *Figure 3 shows the construction process of z-nodes. The q-node Q_3 points to* $\mathrm{UL}(Q_3)$ *of four inter-node trajectories, u_5, u_6, u_7, u_8. To obtain the z-ordering, the space of Q_3 is partitioned based on the start points of the trajectories, and each partition is assigned a z-id where a partition can have at most $\beta = 2$ start points (Figure 3(b)). As an example, the start points of both u_5 and u_6 have* 0.0 *as their z-ids. Next, we apply the same partitioning strategy on the end points. The end points of u_5, u_6, u_7, and u_8 are assigned z-ids* 1.0, 1.2, 2.0, *and* 1.3, *respectively and the partitioning terminates. Finally, a pair of z-ids for each trajectory is kept in z-nodes, each of size β (Figure 2 (right)).*

3.1 Generalization of the Index

So far we have explained our index for trajectories with two points (source and destination), which can only serve a subset of query scenarios. To serve other types applications that require maintaining a sequence of points in each trajectory where a trajectory can be served partially, we generalize our index by proposing two approaches: a segmented approach, and a full-trajectory approach.

Segmented approach. We segment each trajectory into a sequence of pairs of points, and then for each pair of points (segment) we apply the same strategies described above. Here, indexing each segment of the trajectories in hierarchy and ordered lists will enable us to calculate the total and the partial score of service (explained later in Section 4. This process is depicted in Figure 4(a).

Full trajectory approach. Some applications require to consider the entire trajectory contiguously as the objective function need to quantify the coverage of a user trajectory served by the facilities. For such applications, we propose a full-trajectory approach, where we store a trajectory in the q-node at the lowest level of the quadtree that fully contains the entire trajectory. In an intermediate q-node, all inter-node trajectories are sorted using z-orders, and in a leaf q-node intra-node trajectories are stored using z-orders (as described previously). This scenario is depicted in Figure 4(b).

The main reason for using a quadtree is that it supports efficient frequent updates. Moreover, since a quadtree partitions the space into disjoint cells, we can apply z-orders to generate unique IDs for the points in a trajectory.

3.2 Index Storage Cost

The space requirement of the hierarchical component of the TQ-tree includes storing the nodes of the quadtree, specifically, $\mathcal{O}(n|E|)$,

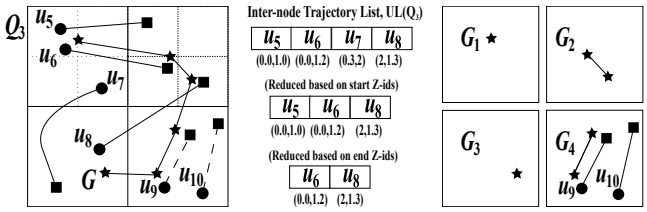


Figure 5: (a) A q-node Q_3 with trajectories and facilities (b) Z-reduce for reducing $UL(Q_3)$ for G (c) Recursive calls for subspaces with corresponding facility subgraphs, G_1, G_2, G_3, G_4.

where n is the number of nodes in the tree and $|E|$ is the expected size of a node. If only the source and destination points of each trajectory is of interest, then a trajectory is stored exactly once in an appropriate node of the TQ-tree. Thus the total size of the user trajectory lists in all nodes, $\sum_{E \in TQ-tree} UL(E)$, is at most the total number of user trajectories $|U|$. The same storage costs apply for the *full trajectory approach* as well.

In the generalized TQ-tree, each segment of a user trajectory is stored in an appropriate node of the TQ-tree. The total number of segments of a trajectory u is $|u|-1$, and a segment is stored exactly once. Thus the total size of the user trajectory lists in all the nodes, $\sum_{E \in TQ-tree} UL(E)$ in the generalized TQ-tree is $\sum_{u \in U} |u|-1$.

3.3 Updating the Index

Since the TQ-tree uses a regular space partitioning scheme, to insert a new user trajectory, u, we can quickly identify the corresponding q-node to which u belongs to in $\mathcal{O}(h)$ time. Then, u needs to be inserted in an appropriate z-node of the user trajectory list. If the number of points in the corresponding z-node does not exceed the threshold β, no further partitioning is needed. The points of u are assigned the appropriate z-ids, and inserted in the sorted user trajectory list. Otherwise, the corresponding z-node is partitioned and the z-ids are assigned to the points of u. Since the z-ids of the existing user trajectories in that z-node may change, we may need to re-assign z-ids to the trajectories. This re-assignment needs to be done for at most β trajectories in that z-node.

4. PROCESSING kBFT QUERIES

In a kBFT query, a user trajectory can be partially served by a facility. Thus, an efficient technique is needed to calculate the appropriate service value of a facility for U. In this section, we first propose an efficient divide-and-conquer algorithm to recursively divide a facility trajectory and traverse only the necessary nodes of the TQ-tree to calculate the service value of the components of the facility in that subspace. We apply a two-phase pruning technique using TQ-tree, where the q-nodes are pruned first, and then the z-orderings are used to further prune the z-nodes. A merge step is evoked to check if the same user trajectory can be served by the connected components of the same facility, and an upper bound of the service value of that facility is updated from the current state of exploration. A best-first strategy is employed to explore the facilities based on their estimated upper bounds of service values. In this section, we first present our algorithm for computing the service value of a single facility $f \in F$. Then we present our approach to find the top-k facilities from F with the maximum service value.

4.1 A Divide-and-Conquer Algorithm

Algorithm 1: evaluateService(Q,f)

Input: A q-node Q of TQ-tree, a facility component f
Output: Service value so of f for users in subtree rooted at Q

1.1 $so \leftarrow 0$
1.2 **if** $f = \varnothing$ **then return** 0
1.3 **if** Q *is a* $leaf$ **then**
1.4 | **return** evaluateNodeTrajectories(Q, f)
1.5 $Q_{children} \leftarrow children(Q)$
1.6 $f_{children} \leftarrow$ intersectingComponents($Q_{children}, f$)
1.7 **for** $q_c \in Q_{children}$, $f_c \in f_{children}$ **do**
1.8 | $so \leftarrow so+$ evaluateService(q_c, f_c)
1.9 $so \leftarrow so+$ evaluateNodeTrajectories(Q, f)
1.10 **return** so

Algorithm 1 shows the pseudocode for the divide-and-conquer algorithm for computing the service value of a facility $f \in F$. Note that in our application scenarios, a user u can be served by f (partially or completely) if a point of u is within a threshold distance ψ from any point of f. Thus, we cover f with an extended minimum bounding rectangle (EMBR) that includes the serving area of f. However, without loss of generality, we use the term *EMBR* and f interchangeably when we match users with f. Initially, the function *evaluateService*$(\cdot)$ is called with the root node Q of the TQ-tree for f. First, it finds the relevant child q-nodes of Q that intersect with f (or EMBR of f) in the function *intersectingComponents*$(\cdot)$ (Line 1.6). If a child q-node does not intersect, that q-node can be safely pruned. Otherwise, the *EMBR* of f is divided into four equal subspaces. For each unpruned child q-node q_c of Q and the corresponding intersecting components of f, the function *evaluateService* in Algorithm 1 is recursively called (Line 1.8).

The recursive call terminates on two conditions: (i) If f is empty (after division there is no point left in that subspace that can server any user (Line 1.2)); and (ii) When Q is a leaf node. For a leaf node, the function *evaluateNodeTrajectories*$(\cdot)$ is called to compute the service value for the intra-node trajectories in $UL(Q)$ of that node.

The function *evaluateNodeTrajectories* is used to determine the service value that is increased for serving the trajectories in *UL*(Q) (Algorithm 2). To check whether the same user trajectory can be served by the same connected components of f, a merge step is employed as the function *MakeUnion*(f) . Here, the connected components of f are assigned unique identifiers. Next, we need to access the trajectories in *UL*(Q) (that are stored as a sorted list of z-nodes according to z-order). We apply a pruning technique using the z-order IDs of the trajectory points to get a list T_r of a reduced size from *UL*(Q) using the *zReduce*$(\cdot)$ function in Lines 2.2 - 2.3 (explained later). For each user trajectory $t_i \in T_r$, we compute the service value gained for serving t_i by f. Note that the evaluation of function *serviceValue*$(\cdot)$ is application specific. For example, for a binary service, *serviceValue*(t_i, f) returns 1 or 0, and for a non-binary service, a normalized distance based service score between $[0, 1]$ can be returned.

The function *zReduce* in Algorithm 2 prunes the inter-node trajectories that cannot contribute to the service value. The idea is to avoid searching the full list of inter-node trajectories and reduce the list to a small relevant set of trajectories based on the spatial properties of f and z-ordering. This function takes the inter node trajectory list and a component of the facility as input. It prunes the user trajectories based on the z-ids that the facility intersects.

EXAMPLE 4. *We explain the zReduce*$(\cdot)$ *function with Figure 5. The figure shows a facility trajectory G, and a list* T_q *of inter-node trajectories* $\{u_5, u_6, u_7, u_8\}$ *with start and end z-ids {(0.0,1.0),*

Algorithm 2: evaluateNodeTrajectories(Q,f)

Input: A q-node Q of TQ-tree, a facility component f
Output: Service value so of f for trajectories stored in Q

2.1 $us \leftarrow$ MakeUnion(f)
2.2 $T_q \leftarrow UL(Q)$
2.3 $T_r \leftarrow$ zReduce(T_q, f)
2.4 $so \leftarrow 0$
2.5 **for** $t_i \in T_r$ **do**
2.6 $\quad so = so +$ serviceValue(t_i, f)
2.7 **return** so

Algorithm 3: TopKFacilities(F, k)

Input: A set of facilities F, a positive integer k
Output: Top k facility collections F'

3.1 Initialize a max-priority queue PQ; $F' \leftarrow \varnothing$
3.2 **if** $F = \varnothing$ **then return** F'
3.3 **for** $f_i \in F$ **do**
3.4 $\quad Q \leftarrow$ containingQNode(f_i)
3.5 $\quad$ *qfPair* $\leftarrow$ makePair(Q, f_i)
3.6 $\quad$ Initialize a state S with id i
3.7 $\quad$ Insert(S.*qflist*, *qfPair*)
3.8 $\quad S$.*aserve* $\leftarrow 0$; S.*hserve* $\leftarrow Q.s_{ub}$
3.9 $\quad$ *fserve*(S) $\leftarrow S$.*aserve* + S.*hserve*
3.10 $\quad$ *PQ*.push(S, *fserve*(S))
3.11 **repeat**
3.12 $\quad S \leftarrow$ *PQ*.pop()
3.13 $\quad$ **if** S.qflist $= \varnothing$ **then**
3.14 $\quad\quad$ Insert($F', S.id$)
3.15 $\quad$ **else**
3.16 $\quad\quad S_{new} \leftarrow$ relaxState(S)
3.17 $\quad\quad$ *PQ*.push(S_{new}, *fserve*(S_{new}))
3.18 **until** $|F'| = k$
3.19 **return** F'

Algorithm 4: relaxState(S)

Input: Current state of a collection of facilities, S
Output: Relaxed state S_r

4.1 Initialize S_r
4.2 **for** *each pair* $(Q, f) \in S$.*qflist* **do**
4.3 $\quad S_r$.*aserve* $\leftarrow S_r$.*aserve* + evaluateNodeTrajectories(Q, f)
4.4 $\quad Q_{children} \leftarrow$ children(Q)
4.5 $\quad f_{children} \leftarrow$ intersectingComponents($Q_{children}$, f)
4.6 $\quad$ **for** $q_c \in Q_{children}$, $f_c \in f_{children}$ **do**
4.7 $\quad\quad$ **if** $f_c \neq \varnothing$ **then**
4.8 $\quad\quad\quad$ *cqgPair* $\leftarrow$ makePair(q_c, f_c)
4.9 $\quad\quad\quad$ Insert(S_r.*qflist*, *cqgPair*)
4.10 $\quad\quad\quad S_r$.*hserve* $\leftarrow S_r$.*hserve* + $q_c.s_{ub}$
4.11 **return** S_r

(0.0,1.2), (0.3,2), (2,1.3)$\}$ *of* Q_3. *Assume,* G *intersects nodes with z-ids* $0.0, 0.1, 1.2, 1.3, 2, 3$ *fully or partially – the stop points in* G *are within* ψ *distance to serve fully or some portions of these z-nodes. Thus, trajectory* u_7 *is pruned since its start z-id* 0.3 *is not covered by* G. *So we get a reduced list* $\{u_5, u_6, u_8\}$ *with z-ids* $\{$*(0.0,1.0), (0.0,1.2), (2,1.3)*$\}$. *Next we look at the z-ids of end points for further pruning. Here,* u_5 *is pruned since its end z-id is* 1.0, *and we get the final reduced list* $\{u_6, u_8\}$. *After reducing* UL(Q) *to* $\{u_6, u_8\}$ *(Figure 5(b)), we divide* G *into four subspaces,* G_1, G_2, G_3, G_4, *and evaluate the service values of these subspaces by calling Function evaluateService*($\cdot$) *(Figure 5(c)).*

Multi-point trajectory processing. The *serviceValue*($\cdot$) function returns a normalized score that is achieved for serving a multi-point trajectory t_i by f (Algorithm 2). The normalized score depends on the requirements of the applications: one may want to count the number of points in u served by f, or find length of the segments of u served by f. To accommodate such applications, the service value calculation changes accordingly.

Based on our above algorithms, we now propose an approach that finds the top-k facilities from a set F of facilities.

4.2 Finding Top-k Facilities

Algorithm 3 shows the pseudocode for finding the top-k facilities. The key idea is to apply a best-first technique to explore facilities based on their predicted service upper bounds. The upper bound of the service value, *fserve* of a facility (or a collection of facilities) is computed by combining the value of the *actual* service function, *aserve*, from the current state of exploration and the optimistic value of the service function, *hserve*, which is estimated based on a *heuristic*, i.e., the maximum service value that can be achieved by further exploration of the facility. For each facility $f_i \in F$, we maintain a tuple S to preserve its current state of exploration. S contains: facility id, a list *qflist* of $\langle$q-node, facility-component$\rangle$ pair that overlaps with each other, the actual service value *aserve* of the facility based on the actual number of users served so far, and the maximum value of the service *hserve* that can be achieved by f_i in the remaining exploration. We maintain a max-priority queue, *PQ* of the tuples S according to the upper bound values, *fserve*, where, *fserve* $=$ *aserve* $+$ *hserve* (i.e., the sum of the actual service value achieved and the upper bound of the service value that can be achieved by the facility).

The result set F' and the priority queue *PQ* is initialized (Line 3.1) as empty. The states are initialized for each facility, and inserted in *PQ* (Lines 3.4 - 3.10). Function *containingQNode*(f_i) returns the smallest q-node, Q that contains f_i (Line 3.4). A pair is formed with the facility component f_i and the corresponding Q. The pair (Q, f_i) is inserted in *qflist* of S. We initialize *aserve* with 0 (as no user trajectories has been matched with f_i) and *hserve* with the upper bound s_{ub} of the service values stored with the node Q in the TQ-tree. As described in Section 3, depending on the application, the upper bound of the service value is different. For example, for scenario 1, s_{ub} of a node Q is the number of trajectories contained in Q. We insert the current state S of f_i along with the total upper bound of the service value, *fserve*(S), achieved so far by the current state of facility exploration.

Next, we progressively explore user trajectories by relaxing different parts of facility trajectories to find the top-k facilities that maximize the service. In each iteration (Lines 3.12 - 3.17), the facility component with the maximum *fserve* value is dequeued from *PQ*, and the state is updated by relaxing the component through a function call *relaxState*(S) that explores the children of the corresponding q-node, updates *aserve* and *hserve*, and inserts the new state into *PQ*. If *qflist* of the dequeued facility is empty, it implies that all components of this facility trajectory are explored, thus the facility is added to the result set. The process terminates when top-k facilities are found, and the result list F' is returned.

State relaxation. Algorithm 4 shows the pseudo-code of how to relax the state of a facility component. The input is the current state, the relaxed (or more expanded) state is returned as output. First, we initialize the variables of the new relaxed state S_r as $S_r.id \leftarrow S.id$,

$S_r.aserve \leftarrow S.aserve$, $S_r.hserve \leftarrow 0$, and $S_r.qflist \leftarrow \varnothing$. Next, for pair (Q, f) of q-node and facility component in the *qflist* of input state S, we expand the component with respect to the children of Q. In this expansion and update process, we update *aserve*, which is the number of users already served by adding the number of inter-node trajectories of the corresponding q-node, that are served by f. For this purpose, we compute the service value of that q-node by calling the *evaluateNodeTrajectories*$(\cdot)$ function, and add this value to $S_r.aserve$, as $S_r.aserve$ denotes the value of trajectories already served (Line 4.3). Next we get the child q-nodes and corresponding components of the facility. In the loop presented in Lines 4.6 - 4.10 we update the list of $\langle$q-node, facility component$\rangle$ pair for each of the child nodes and the maximum value of service $S_r.hserve$ with the upper bound service value s_{ub} stored in the child q-nodes (Line 4.10). The outer loop terminates when we complete the computation for all members of (Q, f) pair list of the current state. Finally the relaxed state S_r is returned.

5. PROCESSING kBCovFT

The kBCovFT query is a variant of the maximum coverage problem, which is NP-hard. A similar problem was presented by Choudhury et al. [8], where given a set of facility locations F, a set of user locations U, and a positive integer k, the problem is to find the top-k facilities from F such that these facilities combinedly serve the maximum number of users from U. In that study, a user is served by a facility if the user is one of the reverse nearest neighbors of that facility, and the problem was shown to be NP-hard. As our problem is very similar to that problem, we omit the proof of NP-hardness from this paper for brevity. Please refer to Lemma 1 in Choudhury et al. for the proof of NP-hardness.

The exact solution of our problem is to iterate through all possible combinations of k facilities from $|F|$ facilities, calculate the service value of each of them, and then return the combination with the maximum value. Although a greedy solution exists with theoretically known best approximation ratio for the maximum coverage problem ([12]), the assumption of the solution is that the objective function is submodular. However, the objective function of the kBCovFT problem is *n*on-submodular, and thus the approximation ratio of that solution does not hold.

LEMMA 1. *The service value function of the kBCovFT problem is non-submodular.*

PROOF. Let $g(\cdot)$ be a function that maps a subset of a finite ground set to a non-negative real number. The function $g(\cdot)$ is submodular if it satisfies the natural "diminishing returns" property: the marginal gain from adding an element x to a set A is at least as high as the marginal gain from adding x to a superset of A. Formally, for all elements x and all pairs of sets $A \subseteq B$, a submodular function satisfies $g(A \cup x) - g(A) \geq g(B \cup x) - g(B)$.

We will prove this lemma by contradiction. Assume that the service function $SO(\cdot)$ of the kBCovFT problem is submodular. Let A be a set of facilities, and $SO(U, A)$ be the maximum number of user trajectories that are combinedly served by A. Now suppose that we add another facility x to A such that $SO(U, A \cup x) = SO(U, A)$ (no additional user is served by adding x). If $SO(\cdot)$ is submodular, then $SO(U, B) \geq SO(U, B \cup x)$ must be true.

If we can find an instance where $SO(U, B) \ngeq SO(U, B \cup x)$ when $SO(U, A \cup x) = SO(U, A)$, $SO(\cdot)$ is non-submodular by contradiction. Consider Scenario 1 where a user u is served by a facility when both the source and destination of u is within ψ distance from any point of the facility. Let the source of a user u be within ψ from a facility in B but not A, and the destination of u is not within ψ from either A or B (u is not served by either). Let the facility x be within ψ distance from only the destination of u. Therefore, u will be served by $B \cup x$ (source is served by B, and destination is served by x). That is, $SO(U, B \cup x) \geq SO(U, B)$. However u is not served by $A \cup x$ as the source of u is not served by A or x, i.e., $SO(U, A \cup x) = SO(U, A)$, which is a contradiction. So, $SO(\cdot)$ of the kBCovFT problem is non-submodular. □

To the best of our knowledge there is no greedy solution with a guaranteed approximation ratio for non-submodular functions for this problem. There are several optimization approaches, including genetic algorithms, simulated annealing, or ant colony optimization that could be used to find the maximum value of the objective function. However, all of these solutions are offline and may require many iterations to converge to an optima, so these solutions are not suitable for the online computation of ad-hoc route planning problems. Therefore, we present a greedy solution of the kBCovFT problem, where the challenge is to efficiently find the users and the user segments that can be combinedly served by multiple facilities, and compute the combined service value, as a user can be served by multiple facilities and there can be overlaps in the service. We exploit the TQ-tree for our solution, as this structure enables us to efficiently address these challenges.

5.1 Greedy Solution

Inspired by the greedy algorithm proposed by Fiege [12], which is the best-possible polynomial time approximation algorithm for the maximum coverage problem, we present a greedy solution for the kBCovFT problem. A straightforward adaptation is to first compute the service value for each facility and iteratively choose a facility that serves the maximum number of users that have not been served, considering the service overlap of multiple facilities for a user. Since this straightforward approach requires to evaluate the services for all facilities and keeping track of all users who have been served by each facility, this approach can be expensive when the number of users and facilities are large.

To overcome the above limitations, we propose a two-step greedy approach, where in the first step we compute a subset ($k' \geq k$) of the highest serving facilities using the kBFT algorithm. In the second step we apply the above mentioned greedy algorithm to iteratively choose a facility from those facilities that serve the maximum number of users that have not been served. We find that this approach is highly effective in practical scenarios and can respond to queries in milliseconds. Due to space constraint, we omit the details, but present its experimental evaluation in Section 7.

6. EXTENSIONS

6.1 Temporal Trajectories

Without loss of generality, our solutions are also applicable for trajectories with temporal information. Our proposed TQ-tree is described for indexing spatial trajectories in Section 3. We can adopt similar concept of hierarchically partitioning the trajectories by clustering the user trajectories based on their time ranges. Let each trajectory, u, consists of a sequence of tuples $u = \{(p_1, t_1), (p_2, t_2), ..., (p_{|u|}, t_{|u|})\}$, where p_1 is the location and t_1 is the timestamp when the user u is located at p_1, and $[t_1 - t_{|u|}]$ is the time range of u. To index time range of all trajectories, we use a binary tree to recursively partition the time window (e.g., 0-24 hrs) into two equal halves, until each leaf node of the tree contains a desired threshold time range (e.g., 1.5 hr). Each node (both internal and leaf) of this temporal tree will index all trajectories whose time range entirely fall in the time span of this node. For example, suppose a branch of the tree contains a leaf node with time

range [9:00-10:29] and another node with time range [9:00-11:59] as its parent. If a trajectory time range is [9:30-10:15], then the trajectory will fall under the leaf node for the corresponding time range [9:00-10:29]; on the other hand if a trajectory time range is [9:15-10:45] then it will fall under the node with time range [9:00-11:59]. A node of the temporal tree manages all trajectories under the node. In particular, each node of the temporal tree, has a pointer to the TQ-tree corresponding to the trajectories that belong to the node. Similarly, a facility query trajectory is represented as a sequence of stop points with expected time range. Thus, the temporal value of the query trajectory is a single time range covering the time ranges of all stop points. To process such a query, we need to check the TQ-tree of the temporal-tree nodes whose time ranges intersect with the query time range. However, if the query time range covers a longer time span, we may need to visit many temporal nodes. Thus, alternatively, we can partition the time range of the query trajectory as multiple smaller time ranges covering different parts of the query trajectory, and then execute these smaller segments of the query trajectory on the index. This helps us to answer the query with reduced number of node access.

6.2 Road Networks

For ease of presentation, we have used Euclidean space to explain our approaches so far. Without loss of generality our proposed approaches also work for road network space. Let $G = (V, E)$ be a road network, V is a vertex set and E is an edge set. Each $v \in V$ represents a road junction or an end of a road, and each edge $e \in E$ is a road segment. Both facility trajectories and user trajectories are map-matched on the spatial networks, and are embedded in the road network graph [1]. When a point of a facility trajectory or a user trajectory map to a vertex then we do not need to alter the road network graph; however, when a facility point or a user point falls on a road segment, then a new vertex is introduced and the corresponding edge is divided into two edges.

Our approach for processing the kBFT query in Euclidean space (Section 4), applies two-phase pruning on the TQ-tree based on a Euclidean distance between a facility trajectory point and a user trajectory point. Since we retrieve user trajectories that are within ψ (Euclidean) distance from a facility point, it is guaranteed that all user trajectories that have road network distance less than or equal ψ are already retrieved. Thus, the algorithm may retrieve some extra candidate trajectories. Therefore, as a last step of the process, we filter those user trajectories whose road network distance from facility points is less than the given threshold using Dijkstra's spatial network expansion approach [9] on the road network graph w.r.t. each facility point.

7. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation for our solutions to answer the kBFT and kBCovFT queries. As there is no prior work that directly answers these problems, we compare our solutions with a baseline. Specifically, for the kBFT query, we compare the following three methods: (i) Baseline (BL): In this approach, for each facility, the user trajectories that are within ψ distance are retrieved by executing a range query in a traditional index (in our experiments, a quadtree). The service value of each facility is computed, and the top-k facilities are returned as the result. (ii) TQ-tree Basic (TQ(B)): In this method, we use a simple TQ-tree that hierarchically organizes user trajectories using a quadtree, but keeps a linear list for storing trajectories in each q-node as the index structure. The algorithm presented in Section 4 is applied on this index. (iii) TQ-tree Z-order (TQ(Z)): We use our proposed TQ-tree, where the trajectories in the hierarchical structure are ordered using a z-curve and indexed using their z-ids in each q-node of the quadtree, and apply the algorithm presented in Section 4. *We present our approach with both TQ(B) and TQ(Z) to show the additional benefits of using the z-ordered bucketing in the index.*

Table 1: Facility trajectory datasets

Name	# Facilities	# of stop points
NY Bus Route	2,024	16,999
Beijing Bus Route	1,842	21,489

Table 2: User trajectory datasets

Name	# Trajectories	Type
NY Taxi-trips (NYT)	1,032,637	point-to-point
NY Foursquare (NYF)	212,751	multi-point
BJ Geolife (BJG)	30,266	multi-point

For the kBCovFT problem, we compare four different methods: (i) Greedy baseline (G-BL) that uses baseline service evaluation strategy in the straightforward greedy approach, (ii) Greedy TQ-tree Basic (G-TQ(B)) that runs our greedy solution using TQ-tree basic, (iii) Greedy TQ-tree Z-order (G-TQ(Z)) using TQ-tree z-order, and (iv) Genetic-TQ-tree Z-order (Gn-TQ(Z)) that employs genetic algorithm using TQ-tree z-order.

7.1 Experimental settings

Algorithms are implemented in Java and ran on a PC equipped with Intel core i5-3570K processor and 8 GB of RAM. In all our experiments, we use in-memory data structures, but can easily be adapted to disk-based solutions.

Facility Datasets. We use two real bus network datasets: (i) New York (NY) and Beijing (BJ) bus routes as our facility datasets. Table 1 shows the summary of the facility datasets. We use subset of facility routes from these real bus networks as our query datasets.

User Trajectory Datasets. To accommodate a wide range of real-world user movements with different types and volumes, we use the following three datasets: (i) Yellow taxi trips[5] in New York (NYT), (ii) Foursquare check-ins[6]in New York (NYF), and (iii) Geolife GPS traces[7] in Beijing (BJG). The taxi-trips are essentially pairs of pick-up and drop-off locations of passengers, and thus can be considered as user trajectories with two points. In contrast, the Foursquare dataset consists of user check-in data for different users in NY, where each check-in is a stop point for a trajectory. We refer these trajectories as *multi-point*. We also use Geolife GPS trajectories that contain the user movement traces of 182 users over three years period of time resulting $30,266$ trajectories in Beijing. This Geolife data can also be considered as *multi-point* user trajectories. Table 2 summarizes the datasets used.

Performance Evaluation and Parameterization. We studied the efficiency, scalability, and effectiveness for the baseline and our proposed approaches by varying several parameters. The list of parameters with their ranges and default values in bold are shown in Table 3. For all experiments, a single parameter is varied while keeping the rest as the default settings.

For efficiency and scalability, we studied the impact of each parameter on (i) the runtime and the number of blocks (i.e., I/Os) accessed while calculating the service value of a facility, and (ii) the

[5]www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

[6]www.kaggle.com/chetanism

[7]www.microsoft.com/en-us/download/details.aspx?id=52367

Table 3: Parameters

Parameters	Ranges
Routes	**NY**, BJ
Datasets	**NYT**, NYF, BJG
# Trajectories	203308, **357139**, 697796, 1032637
# Stops (S)	8, 16, **32**, 64, 128, 256, 512
# Facilities (N)	16, 32, **64**, 128, 256, 512
k	4, **8**, 16, 32

total runtime and the number of block accessed of answering the kBFT query. Since we use in-memory data structures in our experiments, we simulate the I/Os as follows. Assume that a memory block B can contain $|B|$ user trajectories, and thus we need a total of $\left\lceil \frac{|T|}{|B|} \right\rceil$ memory blocks to store $|T|$ trajectories. For each trajectory, the TQ-tree keeps the pointer to a memory block where the original trajectory is stored. Then we count the number of memory blocks that are accessed by each approach while evaluating the queries. In our experiments, $|B|$ is set to 128 by default, assuming that each trajectory is 32 bytes and a disk block is 4096 bytes.

We evaluate the performance on the user trajectory dataset with both source-destination points, and multiple points. In each case we generate 100 sets of queries with the same settings and report the average performance. For each set of query generation, we select N number of facilities with required criteria (e.g., number of stops) from a randomly chosen spatial region of the facility dataset.

As the greedy solutions provide an approximate result, we also report the effectiveness of our solutions as (i) the total number of users served, and (ii) approximation ratio.

7.2 Experimental results

Computing the service value. We vary different parameters and present the processing time & I/O cost for calculating the service value of a single facility in the following.

(i) No. of user trajectories: We vary the number of taxi trips in NYT dataset as 203,308 (NYT-0.5), 357,139 (NYT-1), 697,796 (NYT-2), and 1,032,637 (NYT-3), which corresponds to the taxi trips in 12 hours, 1 day, 2 days, and 3 days, respectively. Figures 6 (a) & (c) show the average processing time and the number of block access, respectively, for the baseline (BL), TQ-tree Basic (TQ(B)), and TQ-tree Z-order (TQ(Z)). As the TQ(B) organizes the trajectory segments in hierarchy in contrast to indexing points in a quadtree in BL, TQ(B) is 1 order of magnitude faster than the baseline. The spatial z-ordering of the trajectories in TQ(Z) results is 2 orders of magnitude faster than TQ(B) for calculating the service value of a single facility. Figure 6 (c) shows that the number of block accesses (which resembles I/Os in a disk based system) in TQ(Z) is at least 2 orders of magnitude less than that of BL, and the number of block accesses in TQ(B) is on average 3.5 times less than that of the baseline.

(ii) No. of facility stops: We vary the number of stops of each facility from 8 to 512, and report the average processing time & I/O cost to compute the service value of a facility. The results (Figure 6 (b)) show that TQ(B) and TQ(Z) outperform the baseline by around 1 order of magnitude and $2-3$ orders of magnitude, respectively. Here, the runtime of all of the approaches gradually increase with the number of stops, as more users become eligible to be served. The benefit of the divide-and-conquer approach in the TQ-tree based approaches is higher for a lower number of stops. Figure 6 (d) shows that TQ(Z) takes at least 2 orders of magnitude fewer block accesses than the baseline and $1-2$ orders of magnitude fewer block accesses than TQ(B).

(iii) Distance threshold ψ: Although more users are likely to be eligible to be served with the increase of ψ, we do not observe any significant change in the performance of our algorithms other than the baseline. We omit the graphs varying ψ for brevity.

Processing kBFT. We evaluate our proposed algorithms for kBFT, and compare the performance with the baseline.

(i) No. of user trajectories: The algorithm using the TQ(Z) index outperforms the baseline by at least $2-3$ orders of magnitude and TQ(B) by around 2 orders of magnitude (Figure 7(a)). As the number of trajectories in the user list of each q-node increases with the total number of user trajectories, the benefits of TQ-tree based indexes decrease gradually. The number of unique z-ids in the z-ordering, and the number of z-nodes also increase with the number of user trajectories, thus the processing time in TQ(Z) increases at a higher rate than the other two approaches. In Figure 8(a) the number of block accesses in TQ(Z) is at least two orders of magnitude less than that of the baseline. However, the TQ(B) requires on average 3.5 times fewer block accesses than the baseline.

(ii) No. of results (k): We vary the number of the required answers k and compare the performance. As the baseline computes the service value of each facility and return k facilities with the maximum values, the processing time of the baseline do not vary for k. The runtime of both TQ-tree based approaches slightly increase with the increase of k as more iterations in the divide-and-conquer approach are likely to be required for a higher k (Figure 7(b)). Similarly, Figure 8(b) shows that the number of block accesses show the similar trends to that of the processing time.

(iii) No. of stops: Similar to the previous results shown for computing the service value of a facility, the processing time when varying the number of stops of each facility gradually increases for all of the approaches (Figure 7(c)). The runtime of TQ(B) is around 1 order of magnitude faster than the baseline for fewer stops, but the benefit decreases as the number of stops increases. The reason is that the number of iterations in the divide-and-conquer approach increases with the number of stops, and the list of trajectories in the user list of a q-node needs to be searched linearly in the TQ(B) each time (as there is no ordering of the trajectories in the list). TQ(Z) consistently outperforms the baseline by around 3 orders of magnitude with the help of the efficient two-level index. Figure 8(c) also shows that TQ(Z) consistently outperforms the baselines in a large margin in terms of number of block access.

(iv) No. of facilities: As more computations are required to find the top-k facilities from a higher number of candidate facilities, the runtime increases for all approaches at around the same rate as shown in Figure 7(d). Although TQ(B) consistently outperforms the baseline, the runtime of the baseline and TQ(B) may not suitable for an efficient ad-hoc route planning with a higher number of facilities. TQ(Z) answers the query on the scale of milliseconds, and is around 3 orders of magnitude faster than the baseline. Similarly, we also observe similar performance trends in all three methods in terms of number of block accesses (Figure 8(d)). We also observe that TQ(Z) requires at least two orders of magnitude fewer block accesses than the BL.

So far we have observed that the processing and the number of block accesses show similar trends in all experimental evaluations. Hence, for brevity we omit the graph of number of block accesses in the remainder of the experiments.

kBFT for multi-point datasets. *NYF Dataset:* Since each user trajectory in the NY Foursquare-check-ins dataset is a sequence of points, we evaluated kBFT queries using the two generalized versions of the index: Segmented TQ-tree (S-TQ) and Full trajectory TQ-tree (F-TQ) (please see Section 3.1). In S-TQ, two consecutive check-ins of a user are considered as a segment, and all such

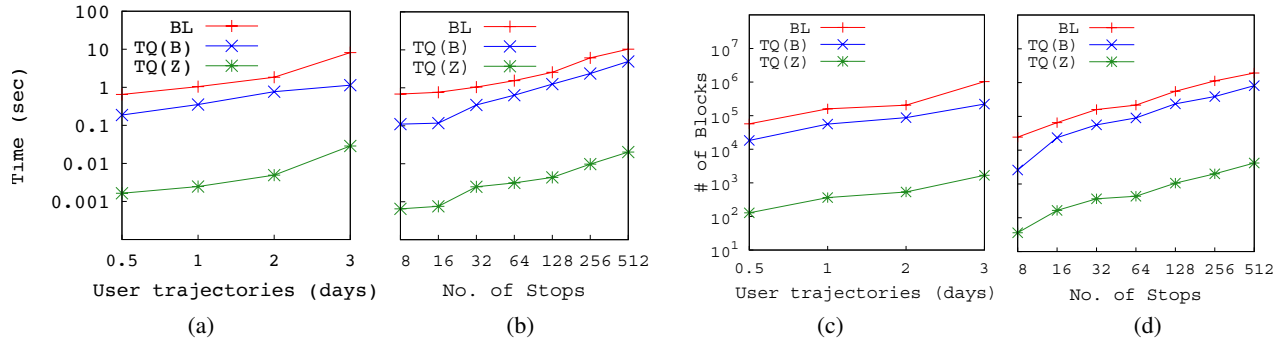


Figure 6: Evaluating service values for varying number of user trajectories (a & c) and stops (b & d) in NYT dataset.

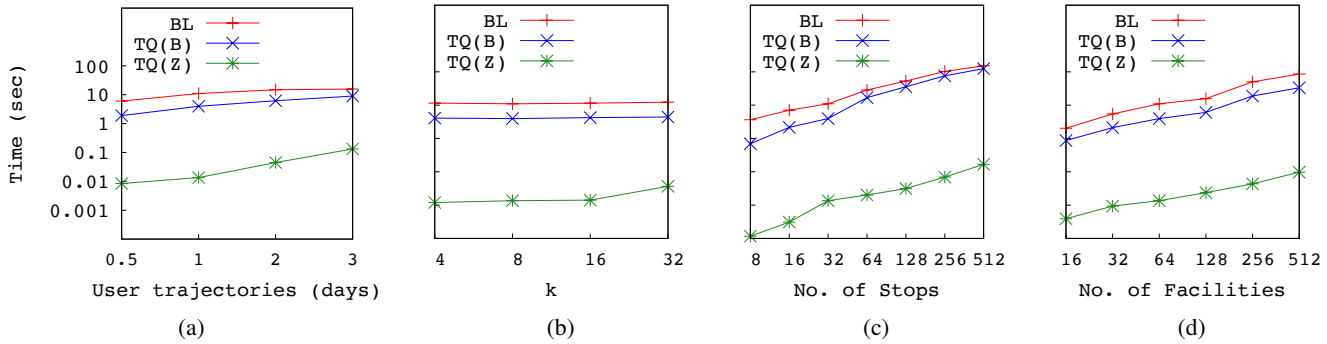


Figure 7: The processing time for evaluating kBFT for varying (a) users, (b) k, (c) stops, and (d) facilities for NYT datasets.

segments of all users are indexed using the TQ-tree. For F-TQ, we consider the sequence of check-ins in a day of a user as a single multi-point trajectory, and index these trajectories using the TQ-tree. For both approaches, we compare the performance for both the TQ-tree basic and the TQ-tree z-order indexes.

Figure 9 shows the results of our approaches when varying (a) the number of stops and (b) number of facilities. The F-TQ based approaches perform better than S-TQ as the number of trajectories increases significantly in the segmented approach. The performance gap between the S-TQ-tree Basic (S-TQ(B)) and the S-TQ(Z) is around 1 order of magnitude, which is smaller than the previous experiments. The underlying reason is that for smaller segments, TQ-tree contains fewer trajectories in the internal nodes and thus z-order based performance gain cannot be achieved. For the same reason, the F-TQ based approaches outperform S-TQ . In all cases, our proposed approaches for processing kBFT using multi-point trajectories significantly outperform the baseline.

BJG dataset: We evaluated our algorithms on another multi-point trajectory dataset from the Geolife project. Since the dataset is small, we run the experiments with the segmented TQ-tree approach, and consider every pair of points as a single trajectory. Figure 10 shows that even for a small dataset our TQ-tree based approaches significantly outperform the baseline.

Evaluating kBCovFT. We evaluated our greedy algorithm, and compare between the competitive approaches. Figure 11 shows the processing time for varying the number of users and facilities for processing kBCovFT in NYT dataset. The G-TQ(Z) outperforms other approaches by a large margin. We also evaluate the quality of our approaches in terms of number of users served (Figure 11(b), Figure 11(d)) and as the approximation ratio with the exact solution (Figure 12). Experimental evaluation shows that the approximation ratio of our greedy TQ(Z) is close to the exact solution in most of the cases, and at least achieves 0.9 ratio. The genetic algorithm (20 iterations) performs poorly in terms of the number of users served when the number of facilities is large (Figure 11(d)).

Table 4: Index size and construction time

# Trajectories	TQ(B): MB (sec)	TQ(Z): MB (sec))
203,308	92 (0.74)	136 (1.03)
357,139	159 (0.95)	237 (1.86)
697,796	301 (2.42)	450 (4.23)
1,032,637	435 (3.74)	655 (9.95)

Index construction and update. In this section we evaluate the index building and updating cost of our proposed index structure.

Memory size and build time: Table 4 shows the index size in MB and the index construction time in seconds for NYT dataset. TQ(Z) takes slightly more space than TQ(B) to store the z-nodes. The index also takes only a few seconds to be built. The overhead for TQ(Z) are marginal when compared to the performance gains achieved by the index.

Index update cost: Our index structure can gracefully handle updates from frequent queries, and has little overall performance impact on the query answer. For example, it takes on average 5 micro-seconds to update a user trajectory in an index of approximately 1 million existing user trajectories. The results show that TQ(Z) takes 3 to 7 times longer and incurs 2 times more I/O than that of TQ(B) while updating the index (graphs are not shown).

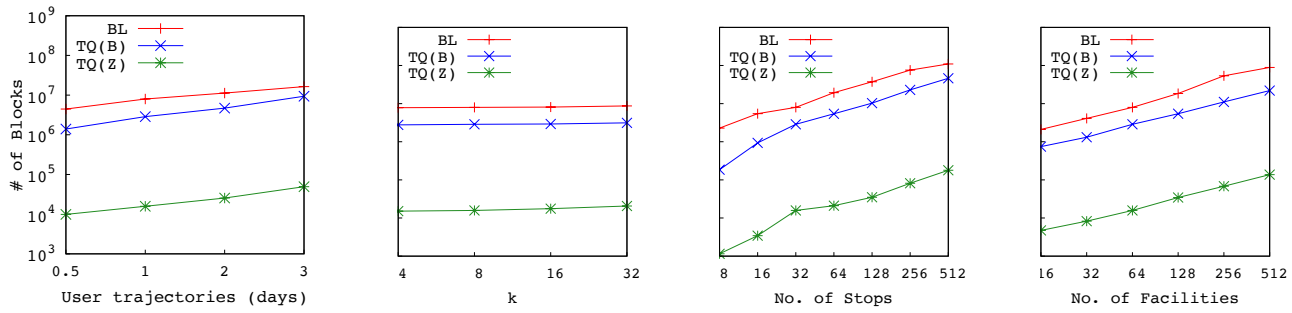


Figure 8: The number of block accesses for evaluating kBFT for varying (a) users, (b) k, (c) stops, and (d) facilities for NYT datasets.

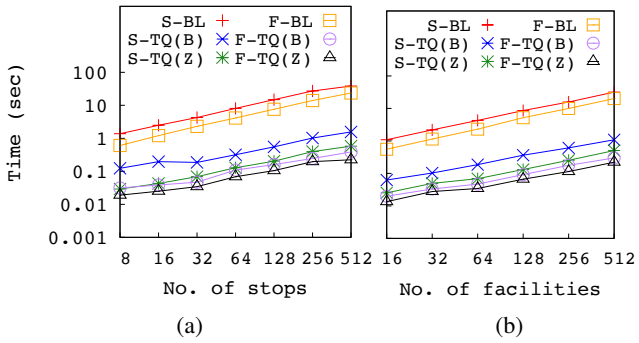


Figure 9: Evaluating kBFT for varying number of (a) stops (b) facilities for New York Foursquare multi-point datasets.

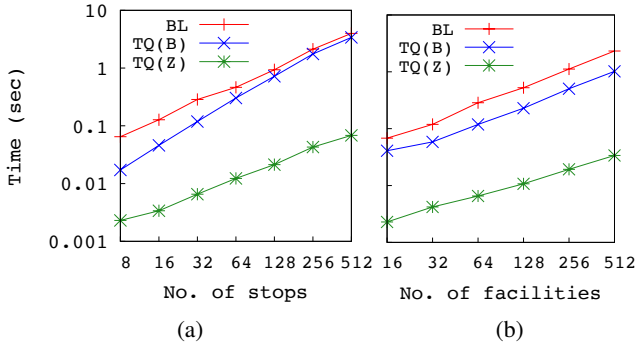


Figure 10: Evaluating kBFT for varying number of (a) stops (b) facilities for Beijing Geolife multi-point datasets.

8. RELATED WORK

The related body of work mostly includes studies in trajectory indexing and query processing, facility location selection problems, and the route planning algorithms.

Trajectory Indexing and Queries. There have been studies for finding human mobility patterns [20, 25] and detecting taxi trajectories [2]. However, as we only use the user trajectories directly as input, the methods for constructing trajectories is outside the scope of this paper. Other relevant studies on trajectories are as follows.

Trajectory Search by Similarity. Frentzos et al. [13] define a dissimilarity metric between two trajectories and apply a best-first technique to return the k most similar trajectories to a query trajectory. Chen et al. [5] address the problem of finding similar trajectories based on the edit distance. A comparative review of different measures of similarity is presented in [30]. Shang et al. [27] study a variant of this problem, where both location and textual attributes of the trajectories are considered. The significance of each point in a query trajectory is taken into consideration in [28], where users can specify a weight for each point in the query trajectory to find the k most similar trajectories using the weights in the similarity function. The general idea is to take each point along the query and check whether a circle with the point as centre and a threshold based on the weight as the radius, touches any trajectory. Based on whether a trajectory covers certain points, a lower and upper similarity bound is calculated, and different pruning techniques are applied. However, this approach is not directly amenable to our problem, as this computation needs to be repeated for each of the facilities, which will incur a high computational cost and unnecessary, repeated retrieval of trajectories. Moreover, this approach cannot efficiently answer the kBCovFT query, where an user trajectory can be served jointly by multiple facility trajectories.

Trajectory Search by Point Location. Given a set of query points, Tang et al. [29] answer the k nearest trajectories, where the distance to a trajectory is the sum of the distances from each query point to its nearest point in that trajectory. Han et al. [16] find the top-k trajectories that are close to the set of query points w.r.t. traveling time. Each of these solutions use R-tree variations to store trajectory points. As computing both the individual and partial service is important in our case, these techniques are not useful for our problem. Adapting these approaches would affect our pruning strategy greatly, resulting in higher computational complexity as it would not be easy to exclude the inter-node trajectories by indexing the points independently. Also, the queries (facilities) in our problem are also trajectories, not just points.

Reverse kNN Trajectory Queries. Given a set of user trajectories U, a set of facility (bus) trajectories F, and a new facility trajectory $f \notin F$, an RkNN query returns the user trajectories from U for which f is one of the k nearest facilities. While addressing this problem, Wang et al. [32] consider each user trajectory as transitions (trajectories with just pickup and drop-off points), and Rahat et al. [23] consider multi-point user trajectories. In contrast to their work, we assume a user can be served by a facility if the trajectories (stop points) are sufficiently close. Moreover, their approach cannot be used to solve the kBCovFT query, where a user trajectory can be served jointly by multiple trajectories.

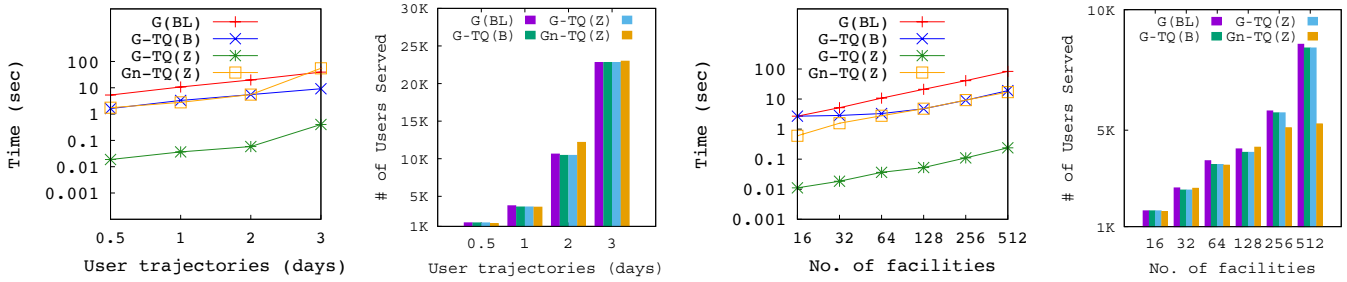


Figure 11: Evaluating kBCovFT for varying (a)-(b) users (c)-(d) facilities for NYT datasets.

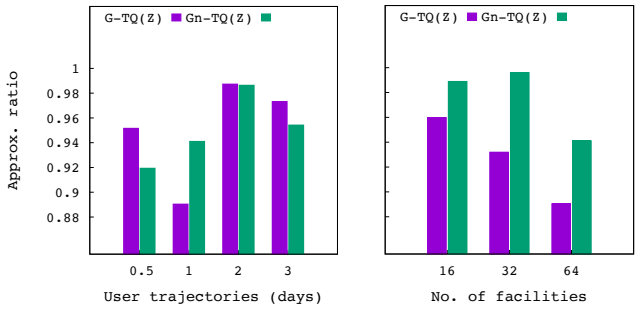


Figure 12: Approximation ratio for evaluating kBCovFT for varying (a) users (b) facilities for NYT datasets.

Other Storage Techniques. Other index structures, e.g., *TrajTree* [24], *SharkDB* [31] are also proposed to efficiently store trajectories. However, as segmentation of trajectories is required to construct TrajTree [24], this index is not amenable to our problem when computing the served portions of the individual user trajectories. SharkDB [31] is an in-memory column oriented timestamped storage solution used for indexing trajectory data. This index can support kNN and window queries in the spatio-temporal domain, but cannot be directly applied to solve kBCovFT where identifying trajectories can be partially served by a facility trajectory.

Distributed Processing of Trajectories. Big trajectory data require distributed framework for efficiency and scalablility. Xie et al. [36] proposed a spatial in-memory big data analytics engine, Simba, to support spatial queries such as kNN, distance join, and join. Simba extends Spark to support spatial queries, and adopts a two-level spatial indexing strategy, composed of local and global indexing, where the global index keeps track of the summarized view of all partitions and the local index accelerates the query processing inside a partition. Later, Xie et al. [35] propose a distributed Spark based framework that support similarity search over trajectories. Ding et al. [10] propose a unified platform for big trajectory data management, namely UlTraMan, that solves several limitations of Spark to handle big trajectory data. They also support distributed computation by using an abstraction called TrajDataset, which is compatible with MapReduce and RDDs. These studies [36, 35, 10] are orthogonal to ours. Interestingly, our proposed TQ-tree based index can be easily adapted for distributed frameworks where the first level TQ-tree as a global index and the second level TQ-trees as local indexes can be used.

Facility Location Selection Problem. Several studies have investigated the problem of finding a location or region to establish a new facility such that the facility can serve the maximum number of customers based on a specified optimization criteria. The facility location problem (FLP) and the optimal location query (OLQ) are the two most popular query types, where in FLP the new facility is selected from given a set of a limited number of possible facility locations [19, 15, 14], and in OLQ, the new facility can be anywhere in the whole space [11, 34, 6]. Another related problem, the *Maximizing Bichromatic Reverse kNN* query [33, 37] finds the optimal region in space for a new facility f such that the number of customers for which f is one of the kNNs, is maximized. These queries focus on point data and thus they are not directly applicable to our problem.

Route/Trip Planning. Bus network design is known to be a complex, non-linear, non-convex, multi-objective NP-hard problem [3]. Based on mobility patterns, there are a number of solutions for recommending driving route [4], discovering popular routes [7], or recommending modification of existing routes/introducing new routes [20]. The MaxRkNNT query [32] focuses on constructing an optimal bus route based on a Reverse kNN trajectory query. Lyu et al. [21] propose new bus routes by processing taxi trajectories while other works [17, 4] aimed at constructing bus routes by analyzing hotspots of user trajectories. In contrast, our proposed queries focus on finding a subset of the query trajectories that serve the highest number of users locally and globally, respectively. So unlike some of the aforementioned works that find the best route offline, we can support online query processing.

9. CONCLUSION

We proposed a novel index structure, the Trajectory Quadtree (TQ-tree) that utilizes a quadtree to hierarchically organize trajectories into different quadtree nodes, and then applies a z-ordering to further organize the trajectories by spatial locality inside each node. We have demonstrated that such a structure is highly effective in pruning the trajectory search space for processing a new class of coverage queries for trajectories: (i) k Best Facility Trajectory Search (kBFT); and (ii) k Best Coverage Facility Trajectory Search (kBCovFT). We have evaluated our algorithms through an extensive experimental study on several real datasets, and demonstrated that our algorithms outperform common baselines by 2 to 3 orders of magnitude. In future work, we will investigate the effectiveness of TQ-tree for other variants of trajectory queries.

Acknowledgement. This work was conducted at DataLab, BUET and partially supported by Australian Research Council DP170102231.

10. REFERENCES

[1] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2005.

[2] C. Chen, D. Zhang, P. S. Castro, N. Li, L. Sun, and S. Li. Real-time detection of anomalous taxi trajectories from GPS traces. In *MobiQuitous*, pages 63–74, 2011.

[3] C. Chen, D. Zhang, N. Li, and Z. Zhou. B-planner: Planning bidirectional night bus routes using large-scale taxi GPS traces. *IEEE Trans. Intelligent Transportation Systems*, 15(4):1451–1465, 2014.

[4] C. Chen, D. Zhang, Z. Zhou, N. Li, T. Atmaca, and S. Li. B-planner: Night bus route planning using large-scale taxi GPS traces. In *PerCom*, pages 225–233, 2013.

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

[6] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks. In *SIGMOD*, pages 123–134, 2014.

[7] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.

[8] F. M. Choudhury, J. S. Culpepper, T. Sellis, and X. Cao. Maximizing bichromatic reverse spatial and textual k nearest neighbor queries. *PVLDB*, 9(6):456–467, 2016.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.

[10] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao. Ultraman: A unified platform for big trajectory data management and analytics. *PVLDB*, 11(7):787–799, 2018.

[11] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.

[12] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

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

[14] Y. Gao, S. Qi, L. Chen, B. Zheng, and X. Li. On efficient k-optimal-location-selection query processing in metric spaces. *Inf. Sci.*, 298(C):98–117, 2015.

[15] Y. Gao, B. Zheng, G. Chen, and Q. Li. Optimal-location-selection query processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(8):1162–1177, 2009.

[16] Y. Han, L. Chang, W. Zhang, X. Lin, and L. Wang. Efficiently retrieving top-k trajectories by locations via traveling time. In *ADC*, pages 122–134, 2014.

[17] Y. Huang, F. Bastani, R. Jin, and X. S. Wang. Large scale real-time ridesharing with service guarantee on road networks. *PVLDB*, 7(14):2017–2028, 2014.

[18] "IEEE". Test autonomous ride sharing. `spectrum.ieee.org/cars-that-think/transportation/self-driving/driveai-partners-with-lyft-for-autonomous-ride-sharing-pilot`. [Online; accessed 17-09-2017].

[19] Q. Jianzhong, Z. Rui, L. Kulik, D. Lin, and X. Yuan. The min-dist location selection query. In *ICDE*, pages 366–377, 2012.

[20] Y. Liu, C. Liu, N. J. Yuan, L. Duan, Y. Fu, H. Xiong, S. Xu, and J. Wu. Exploiting heterogeneous human mobility patterns for intelligent bus routing. In *ICDM*, pages 360–369, 2014.

[21] Y. Lyu, C. Chow, V. C. S. Lee, Y. Li, and J. Zeng. T2CBS: mining taxi trajectories for customized bus systems. In *INFOCOM*, pages 441–446, 2016.

[22] S. Ma, Y. Zheng, and O. Wolfson. T-share: A large-scale dynamic taxi ridesharing service. In *ICDE*, pages 410–421, 2013.

[23] T. A. Rahat, A. Arman, and M. E. Ali. Maximizing reverse k-nearest neighbors for trajectories. In *ADC*, pages 262–274, 2018.

[24] S. Ranu, D. P, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *ICDE*, pages 999–1010, 2015.

[25] S. Shafique and M. E. Ali. Recommending most popular travel path within a region of interest from historical trajectory data. In *MobiGIS*, pages 2–11, 2016.

[26] S. Shang, L. Chen, Z. Wei, C. S. Jensen, K. Zheng, and P. Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[27] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *EDBT*, pages 156–167, 2012.

[28] S. Shang, R. Ding, K. Zheng, C. S. Jensen, P. Kalnis, and X. Zhou. Personalized trajectory matching in spatial networks. *VLDB J.*, 23(3):449–468, 2014.

[29] L. A. Tang, Y. Zheng, X. Xie, J. Yuan, X. Yu, and J. Han. Retrieving k-nearest neighboring trajectories by a set of point locations. In *SSTD*, pages 223–241, 2011.

[30] H. Wang, H. Su, K. Zheng, S. W. Sadiq, and X. Zhou. An E ectiveness study on trajectory similarity measures. In *ADC*, pages 13–22, 2013.

[31] H. Wang, K. Zheng, X. Zhou, and S. W. Sadiq. Sharkdb: An in-memory storage system for massive trajectory data. In *SIGMOD*, pages 1099–1104, 2015.

[32] S. Wang, Z. Bao, J. S. Culpepper, T. K. Sellis, and G. Cong. Reverse k nearest neighbor search over trajectories. *CoRR, abs/1704.03978*, 2017.

[33] R. C. Wong, M. T. Özsu, P. S. Yu, A. W. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.

[34] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.

[35] D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *PVLDB*, 10(11):1478–1489, 2017.

[36] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.

[37] Z. Zhou, W. Wu, X. Li, M. Lee, and W. Hsu. Maxfirst for maxbrknn. In *ICDE*, pages 828–839, 2011.

[38] C. Zhu, J. Xu, C. Liu, P. Zhao, A. Liu, and L. Zhao. Efficient trip planning for maximizing user satisfaction. In *DASFAA*, pages 260–276, 2015.