Towards Efficient Search for Activity Trajectories

Zheng, Kai; Shang, Shuo; Yuan, Jing; Yang, Yi

Published in:
Proceedings of the 29th IEEE International Conference on Data Engineering

DOI (link to publication from Publisher):
10.1109/ICDE.2013.6544828

Publication date:
2013

Document Version
Early version, also known as pre-print

Towards Efficient Search for Activity Trajectories

Kai Zheng[1], Shuo Shang[2], Nicholas Jing Yuan[3], Yi Yang[4]

[1] School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane 4072, Australia, kevinz@itee.uq.edu.au
[2] Department of Computer Science, Aalborg University, sshang@cs.aau.dk
[3] Microsoft Research Asia, Beijing, China, nichy@microsoft.com
[4] School of Computer Science, Carnegie Mellon University, PA, USA, yiyang@cs.cmu.edu

***Abstract*—The advances in location positioning and wireless communication technologies have led to a myriad of spatial trajectories representing the mobility of a variety of moving objects. While processing trajectory data with the focus of spatio-temporal features has been widely studied in the last decade, recent proliferation in location-based web applications (e.g., Foursquare, Facebook) has given rise to large amounts of trajectories associated with activity information, called activity trajectory. In this paper, we study the problem of efficient similarity search on activity trajectory database. Given a sequence of query locations, each associated with a set of desired activities, an activity trajectory similarity query (ATSQ) returns k trajectories that cover the query activities and yield the shortest minimum match distance. An order-sensitive activity trajectory similarity query (OATSQ) is also proposed to take into account the order of the query locations. To process the queries efficiently, we firstly develop a novel hybrid grid index, GAT, to organize the trajectory segments and activities hierarchically, which enables us to prune the search space by location proximity and activity containment simultaneously. In addition, we propose algorithms for efficient computation of the minimum match distance and minimum order-sensitive match distance, respectively. The results of our extensive empirical studies based on real online check-in datasets demonstrate that our proposed index and methods are capable of achieving superior performance and good scalability.**

I. Introduction

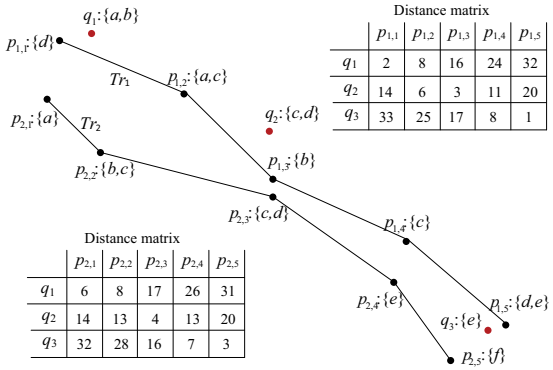
Driven by major advances in sensor technology, GPS-enabled mobile devices and wireless communication, a large amount of data describing the motion history of moving objects, known as *trajectory*, are currently generated and managed in scores of application domains. This inspires tremendous efforts made on analyzing large scale trajectory data from a variety of aspects in the last decade. Representative work includes designing effective trajectory indexing structures [1][2][3][4][5], efficient trajectory query processing [6][7][8], uncertainty management [9][10], and mining knowledge/patterns from trajectories [11][12][13][14][15], to name a few.

In spite of the significant contributions made by this work, they mainly focus on the spatio-temporal features of the trajectories. Typically, a trajectory is modelled as a sequence of time-stamped geo-locations in two or three dimensional space, which means spatio-temporal information is essential to a trajectory database. However, recent years have witnessed the flourish of location-based web applications such as online check-in services (e.g., Foursquare[1]), location or route sharing (e.g., Facebook Place[2], Bikely[3]) and geo-tagged media sharing (e.g., Flickr[4]). These applications are redefining and enriching the traditional trajectory databases by associating locations with semantic meanings. For example, Foursquare users can check in the venues they are visiting and leave tips for other people. Flickr allows tourists to upload their geo-tagged photos took by smartphones during the travel, so that their trips can be outlined by the time and location information embedded in the photos. From the trajectories generated in these applications, we can know not only where and when a user has been, as in the traditional trajectory database, but also what he/she has done by extracting the information from the multimedia contents attached to the locations (e.g., text, images, videos).

In this paper, we use the term *activity trajectory* to represent this new type of trajectory data that contains the information about the user activities at particular places. Tr_1 and Tr_2 in Figure 1 exemplify the activity trajectories, where each place is associated with a set (could be empty) of activities performed by the users. Notice that how to extract and classify the activities is orthogonal to the techniques in our paper, and we just regard each activity as a unique entry of a pre-defined activity vocabulary. Since the activity trajectories are becoming ubiquitous and still growing in a fast pace, analyzing them is undoubtedly an important problem which will lead to many interesting findings. Towards this direction, we study the problem of efficient similarity search in large scale activity trajectories. Similarity search has been studied for long in trajectory databases due to its broad range of real applications. Consequently, a great number of research results exists including different similarity/distance measures and search techniques [16][17][18][6][7][19][20]. But none of these methods can be applied to activity trajectories, as the activities must be considered in both distance measures and search process.

Identifying the similar activity trajectories to a given query is very useful for place recommendation and trip planning. Consider the example in Figure 1 where a tourist plans to visit three places q_1, q_2, q_3 in a city and conduct activities $\{a, b\}, \{c, d\}, \{e\}$, respectively. Because he is not familiar with

[1] http://www.foursquare.com
[2] http://www.facebook.com
[3] http://www.bikely.com
[4] http://www.flickr.com



	$p_{1,1}$	$p_{1,2}$	$p_{1,3}$	$p_{1,4}$	$p_{1,5}$
q_1	2	8	16	24	32
q_2	14	6	3	11	20
q_3	33	25	17	8	1

	$p_{2,1}$	$p_{2,2}$	$p_{2,3}$	$p_{2,4}$	$p_{2,5}$
q_1	6	8	17	26	31
q_2	14	13	4	13	20
q_3	32	28	16	7	3

Fig. 1: Similarity query example

this city, he would like to look at the travelling histories of other people nearby his intended locations for reference. If only the geometric property of the trajectories is to be considered, we can apply the *best match distance* measure proposed by Chen et al [20], which aims at searching for similar trajectories with respect to multiple query locations. By doing so, Tr_1 will be taken as the most promising result since its points $p_{1,1}, p_{1,3}, p_{1,5}$ are closer to the query points than Tr_2. But obviously the tourist will not be satisfied by this result since $p_{1,1}, p_{1,3}$ do not cover his intended activities. On the other hand, Tr_2 can be a better reference for the tourist as it has the points around each query location while matching the activities ($p_{2,1}, p_{2,2}$ for q_1, $p_{2,3}$ for q_2, $p_{2,5}$ for q_3), though it is a bit further than Tr_1 from pure geographical aspect.

Having observed the limitation of traditional trajectory similarity search, we propose a novel similarity query for activity trajectories by incorporating both geometric distance and activity match into the similarity measure, with the goal of returning more meaningful results to the users. However, answering this new query turns to be a more challenging problem since just making use of either location or activity information for pruning the search space will result in bad query performance. Our approach to this problem starts with a novel grid index called *GAT*, which includes a hierarchy of cells for each activity, an inverted list of trajectories containing each activity within each cell, and a summarized sketch of activities for each trajectory. GAT keeps the advantage of hierarchical spatial index like R-tree [21] while avoiding the flaws of large "dead zones" when indexing trajectories by minimum bounding boxes. In addition, the index not only uses the local information on trajectory segments within the cells but also preserves some global information for the entire trajectory in the activity sketch, so that its pruning power can be boosted. On top of the index, we develop a best-first search strategy with tighter distance lower bound for all "unseen" trajectories in the database and an efficient algorithm to compute the distance between candidates and the query. Furthermore we extend the similarity query to be"order-sensitive" by taking into account the order of the query points, and propose efficient solutions to rule out invalid candidates and evaluate the more costly distance function. To sum up, we make the following major contributions in this paper.

- We introduce and formalize two new types of similarity queries for activity trajectories.
- We propose a novel grid indexing structure called *GAT* to organize the trajectory segments and their activity information in a hierarchical manner. On top of that, we develop a best-first search framework, which consists of candidate retrieval and validation procedures, to prune a large number of disqualifying trajectories by spatial proximity and activity containment simultaneously.
- We also develop efficient algorithms to compute the minimum (order-sensitive) match distance between a query and a candidate trajectory.
- We conduct an extensive experimental study based on real trajectory datasets, which includes performance comparisons with three baseline algorithms and memory cost evaluation on the proposed index. The experimental results demonstrate the efficiency and scalability of our proposed solution.

The remainder of the paper is organized as follows. We define the necessary concepts and formulate the similarity query in Section II. Section III presents the baseline methods. Proposed indexing structure and solution for ATSQ are discussed in Section V. Section VI defines the OATSQ and describes our approach. Section VII reports the experimental observations, followed by a brief review of related work in Section VIII. Section IX concludes the paper.

II. Problem Statement

In this section, we give the problem statement and provide necessary definitions and background. Table I summarizes the notations used throughout the paper.

Definition 1 (Activity): An activity α represents a type of action that a user can take at some place of interest such as sport, dining and entertaining. We use $\mathbb{A}$ to denote the pre-defined activity vocabulary, which is the union of all the activities the can be performed by the users.

Definition 2 (Activity Trajectory): An activity trajectory Tr is defined as a sequence of geo-spatial points associated with activities, i.e., $Tr = (p_1, p_2, ..., p_n)$. Each p_i represents a geo-spatial location, which is attached with a (possibly empty) set of activities $\Phi \subseteq \mathbb{A}$.

Essentially, an activity trajectory is historical record describing what a user did and where he/she did it. In the rest of the paper, we will simply use *trajectory* to represent *activity trajectory* when no ambiguity can be caused.

Definition 3 (Point Match): Given a query point q with a set of activities $q.\Phi$, a point match from Tr to q, denoted by $Tr.PM(q)$, is a set of points $P \subseteq Tr$ such that its union of activities is a superset of $q.\Phi_i$, i.e., $q.\Phi \subseteq \bigcup_{p_i \in P} p_i.\Phi$. The sum distance between each point in $Tr.PM(q)$ and q, i.e., $D_{pm}(q, Tr.PM(q)) = \sum_{p \in Tr.PM(q)} d(p, q)$, is called the point match distance.

TABLE I: Summary of notations

Notation	Definition
Tr	Activity trajectory
α	An activity
p	A point in trajectory
$p.\Phi$	The set of activities attached to p
$Tr[i,j]$	Sub-trajectory of Tr from p_i to p_j
Q	A set of query locations
q	A query location in Q
$Tr.PM(q)$	A point match from Tr to q
$Tr.MPM(q)$	The minimum point match from Tr to q
$Tr.M(Q)$	A match from Tr to Q
$Tr.MM(Q)$	The minimum match from Tr to Q
$Tr.OM(Q)$	An order-sensitive match from Tr to Q
$Tr.MOM(Q)$	The minimum order-sensitive match from Tr to Q
$D_{pm}(q, Tr.PM(q))$	Point match distance from $Tr.PM(q)$ to q
$D_{mpm}(q, Tr)$	The minimum point match distance from Tr to q
$D_m(Q, Tr.M(Q))$	The match distance from $Tr.M(Q)$ to Q
$D_{mm}(Q, Tr)$	The minimum match distance from Tr to Q
$D_{om}(Q, Tr.M(Q))$	The order-sensitive match distance from $Tr.OM(Q)$ to Q
$D_{mom}(Q, Tr)$	The minimum order-sensitive match distance from Tr to Q

Obviously, there may be none or multiple point matches in Tr for a given query point. So we define the concept of minimum point match.

Definition 4 (Minimum Point Match): Given a query point q and a trajectory Tr, a point match $Tr.PM(q)$ (if exists) is called the minimum point match, denoted as $Tr.MPM(q)$, if for any other point match $Tr.PM(q)'$, we have $D_{pm}(q, Tr.PM(q)) \leq D_{pm}(q, Tr.PM(q)')$. $D_{pm}(q, Tr.PM(q))$ is called the minimum point match distance between Tr and q, denoted as $D_{mpm}(q, Tr)$.

Considering the example in Figure 1, any point set in Tr_1 that is the superset of any of the four point sets, namely $\{p_{1,1}, p_{1,2}\}$, $\{p_{1,1}, p_{1,4}\}$, $\{p_{1,2}, p_{1,5}\}$, $\{p_{1,4}, p_{1,5}\}$, is a point match from Tr_1 to q_2. With the distance matrix shown in the figure, $\{p_{1,1}, p_{1,2}\}$ is the minimum point match.

Intuitively, the minimum point match is the set of points in the trajectory that collectively meet the activity requirement and have the closest distance to the query point. Next we extend this concept to multiple query points.

Definition 5 (Match): Given a set of query locations Q : $(q_1, q_2, ..., q_m)$, we say a trajectory Tr is a match to Q, denoted as $Tr.M(Q)$, if the point match exists for each query point $q_i \in Q$. The set of point matches for each query point forms the match from Tr to Q. Besides, the sum of point match distances is called the match distance, i.e., $D_m(Q, Tr.M(Q)) = \sum_{q \in Q} D_{pm}(q, Tr.PM(q))$.

Definition 6 (Minimum Match): Given a set of query locations Q and a trajectory Tr, a match $Tr.M(Q)$ is called the minimum match, denoted as $Tr.MM(Q)$, if for any other match $Tr.M(Q)'$ we have $D_m(Q, Tr.M(Q)) \leq D_m(Q, Tr.M(Q)')$. $D_m(Q, Tr.M(Q))$ is called the minimum match distance between Q and Tr, denoted as $D_{mm}(Q, Tr)$.

Continuing the above example, we can get that the minimum match from Tr_1 to Q is $\{\{p_{1,2}, p_{1,3}\}, \{p_{1,1}, p_{1,2}\}, \{p_{1,5}\}\}$, and $Tr_2.MM(Q) = \{\{p_{2,1}, p_{2,2}\}, \{p_{2,3}\}, \{p_{2,4}\}\}$. By this way, Tr_2 is considered to be more similar to the query than Tr_1 based on the their minimum match distances. The following lemma states the relationship between the minimum point match and minimum match.

Lemma 1: Given a query Q and a trajectory Tr, the minimum match $Tr.MM(Q)$ is formed by the minimum point match for each query point $q_i \in Q$, i.e., $\{Tr.MPM(q_1), Tr.MPM(q_2), ...\}$. The minimum match distance is the sum of minimum point match distance, i.e., $D_{mm}(Q, Tr) = \sum_{q_i \in Q} D_{mpm}(q_i, Tr)$.

The proof is omitted due to the space limit. According to this lemma, finding the minimum match for a query Q can be decomposed into looking for the minimum point match for each point in Q.

Activity Trajectory Similarity Query (ATSQ). Given an activity trajectory set $\mathcal{D}$, a query Q, a positive integer k, an Activity Trajectory Similarity Query (ATSQ) returns k distinct trajectories from $\mathcal{D}$ that have the smallest minimum match distances with respect to Q.

Ideally, the ATSQ will return the trajectories that contain the query activities at the places close to each query location. Here to offer some degree of flexibility, the order of the query locations is not considered. But we will extend the query definition to be order-sensitive in Section VI which, as we shall see later, makes the query processing more complicated since the Lemma 1 will not hold.

III. Baseline Algorithms

No baseline method exists for the ATSQ. In this section, we propose three baseline algorithms which explore the possibility of using existing techniques to solve this problem.

A. Inverted List based algorithm

The first baseline algorithm, called IL, only utilizes the activities to prune the search space. Specifically, it aggregates the activities associated with each point in a trajectory, and then builds an inverted list for each activity. The basic idea is to firstly filter out the trajectories in database that do not contain all the activities specified in the query. Then for the remaining candidates, we will sequentially process each of them to compute the minimum match distance with respect to the query, and then return the top-k results.

B. R-tree based algorithm

The second baseline method uses R-tree [21] as the indexing structure to prune the search space in pure spatial dimension. Firstly we treat the points of all trajectories as a point set and index these points using an R-tree. Given a query Q, the baseline traverses R-tree to find the nearest trajectory incrementally in terms of the *best match distance*. The best match distance

(D_{bm}) between a query Q and a trajectory Tr is defined as the sum distances from each query point in Q to its nearest point in Tr, i.e., $D_{bm}(Q, Tr) = \sum_{q \in Q} minDist(q, Tr)$, where $minDist(q, Tr) = \min_{p \in Tr} d(q, p)$.

It is easy to see that the best match distance always lower bounds the minimum match distance, formally presented by the following lemma.

Lemma 2: $D_{bm}(Q, Tr) \leq D(Q, Tr)_{mm}$

We can adapt the algorithm proposed in [20], which is designed to answer the k-BCT query efficiently. Whenever the next nearest trajectory is retrieved, we compute its minimum match distance if it is a match with respect to Q. During the process, we keep track of the k-th minimum match distance as the threshold. If the best match distance of the next obtained candidate exceeds this threshold, the algorithm can terminate since it is guaranteed that all "unseen" trajectories will not have the minimum match distance smaller than the current top-k results, due to Lemma 2.

C. IR-tree based algorithm

The third baseline adopts the IR-tree [22] as the indexing structure, which is used to support efficient spatial keyword search on static point set. The IR-tree is essentially an R-tree extended with inverted files [23]. Each leaf node in the IR-tree contains a number of entries with the form $(p, p.r, p.if)$, where p refers to the pointer of a spatial object, $p.r$ is the bounding rectangle of p, and $p.if$ is a pointer to an inverted file for the text descriptions of the objects stored in this node. Each non-leaf node R contains a number of entries of the form $(cp, cp.rect, cp.if)$ where cp is the address of a child node of R, $cp.rect$ is the MBR of all the rectangles in the entries of child nodes, and $cp.if$ is a pointer to an inverted file for the union of the text descriptions of its child nodes. The search algorithm based on IR-tree proceeds similarly with the R-tree based method, i.e., trying to find the most spatially close trajectories and then computing the minimum match distance with respect to the query. The only modification is, before probing the entries in a node of IR-tree, we first check its inverted file to see if it contains any activity of the query. If not, all the places enclosed in this node can be pruned directly. By this means, this baseline is expected to examine fewer nodes than the R-tree based method, thus can achieve better efficiency.

IV. Proposed Indexing Structure

In this paper, we propose a novel Grid index for Activity Trajectories (GAT). Specifically, we construct a d-Grid by dividing the entire spatial region into $2^d \times 2^d$ quad cells. Then we further build $(d-1)$-Grid,$(d-2)$-Grid,...,1-Grid, which will form a hierarchy of cells. Each cell can be assigned a unique numerical ID by using space filling curve, which maps multidimensional cells to 1-dimensional integer domain. As shown in Figure 2(a), cell 1 to cell 16 form the 2-Grid, and cell 17 to cell 20 form the 1-Grid. In addition, GAT consists of the following four components, which are illustrated in Figure 2(b).

Hierarchical Inverted Cell List (HICL). To facilitate identifying the regions that contain the query activities, we firstly build an inverted list of cells in the d-Grid for each activity α existing in the dataset. After that, we aggregate the cells that belong to the same parent cell in the $(d-1)$-Grid to build a higher-level inverted cell list. By repeating the above process until reaching the 1-Grid, we eventually build a hierarchical inverted cell list for each activity. Next, we discuss the storage of this structure.

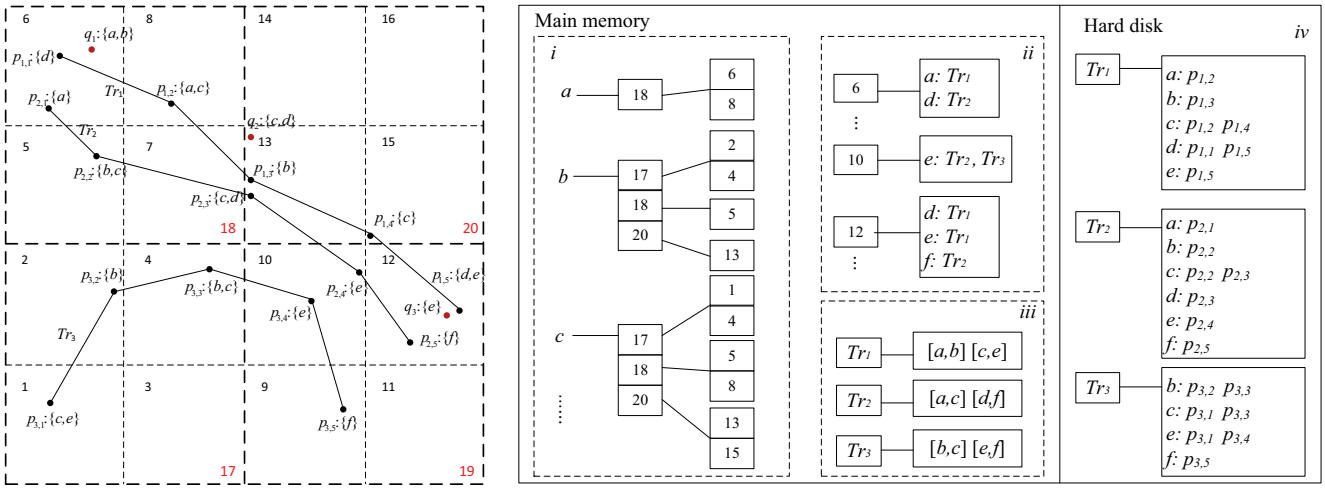
Since the number of activities can be very large, maintaining the entire HICL in the main memory may become infeasible. In this case, we can just keep the high levels of the structure within main memory and the low levels on the secondary storage. More specifically, given a memory budget B, and the cardinality of activity vocabulary C, we can estimate the level h, the levels higher than which will be put on the secondary storage, by choosing the largest integer h satisfying $\sum_{i=1}^{h} 4^i C \leq B$, i.e., $h = \lfloor \log_4(\frac{3B}{4C} + 1) \rfloor$.

Inverted Trajectory List (ITL). In each cell of the d-Grid, we build an inverted trajectory list for each activity α existing in this cell, which is a list of trajectory IDs whose segment contains α within this cell. This structure provides the activity information on the trajectory level, i.e., which trajectories contain α in this cell. Since it does not keep the detailed information about individual points for each trajectory, the size of ITL is much smaller compared to the original dataset, and hence ITL can be accommodated within the main memory of a mainstream server in most cases. However, in the circumstances where only limited memory is available, we can partition all the cells into fewer but big blocks, save all the blocks in the secondary storage, and then retrieve the block(s) around the query location into main memory at query time.

Trajectory Activity Sketch (TAS). For each trajectory Tr in the database, we build an *activity sketch* in the main memory, which summarizes the activities contained in Tr by using relatively small memory space. The purpose of this data structure is to quickly filter out the trajectories whose activities do not match the query requirement without retrieving all the detail information from the disk.

To this end, we sort all the activities in the vocabulary by their occurrence frequencies in the whole database, and assign continuous numerical ID to each activity. Then for each trajectory Tr, we partition its activity IDs into M intervals, $\mathcal{I} = \{I_1, I_2, ..., I_M\}$, with the goal of minimizing the overall size of the intervals, i.e., $\sum_{a=1}^{M} |I_a|$, where $|I_a|$ is defined as the difference between the greatest and smallest IDs in I_a. The reason of doing this is to make the intervals as compact as possible so that the pruning effect can be maximized. The choice of M can be made according to the memory budget. Large M (i.e., more intervals) is expected to gain better pruning effect. Since each interval only needs to keep two integers (which cost 8 bytes), the total memory cost for N trajectories is $8MN$ bytes.

To derive the desired partition, we first sort the activity IDs for each trajectory, and then compute the gap between



(*a*) Grid partition of space

(*b*) Components of grid index

Fig. 2: Grid index for activity trajectories, with illustration of the four components: (i) Hierarchical inverted cell list. (ii) Inverted trajectory list. (iii) Trajectory activity sketch. (iv) Activity posting list.

consecutive IDs. Finally the top $M-1$ largest gaps are chosen as the split positions to partition all activities into M intervals. It is not difficult to prove this is the optimal partition, since relocating any split point (with gap g) to other places (with gap g') will result in increase by $g - g'$ on the overall size of the intervals ($g > g'$).

Activity Posting List (APL). For each trajectory Tr in the database, we construct an activity posting list for each activity α existing in Tr, which is a list of the trajectory points that contain α. This data structure is stored on disk due to its high space requirement, and will be retrieved only when the distance with the query needs to be evaluated.

V. Proposed Search Algorithm

The basic structure of our proposed search algorithm is illustrated by Algorithm 1. Firstly, we retrieve a set of λ candidate trajectories, which contains some places nearby any of the query locations and at least one of the query activities. The second step is to validate each candidate if it is a whole match with respect to the query. Finally we compute the minimum match distance for each valid candidate and insert it into the result set. During this process, we keep track of k-th smallest minimum match distance (MMD^k_{mm}) found so far and a *lower bounding distance* (D_{lb}) for all "unseen" trajectories. As long as $D^k_{mm} < D_{lb}$, the algorithm can terminate safely since all the "unseen" trajectories are impossible to become the top-k results. Otherwise we will incrementally fetch more candidates and repeat the above process again.

A. Candidate Retrieval

A candidate is a trajectory that is possible and seemingly promising to become a result for the query. Since the query trajectory can consist of several places spanning a large area, we will obtain a set of candidates which are close to at least one of the query locations and contain at least one of the query activities at that location.

Algorithm 1: Search Algorithm Outline

Input: trajectory database $\mathcal{D}$, query Q
Output: top-k result set $\mathcal{R}$

```
while true do
    CS ← ∅;
    CS ← retrieve at least λ new candidates;
    D_lb ← update the lower bounding distance;
    for each Tr ∈ CS do
        if Tr is a valid candidate then
            Compute D_mm(Q, Tr) and put Tr into R;
            Update D^k_mm and D_lb;
    if D^k_mm < D_lb then
        break;
Keep the topk results in R;
return R;
```

We adapt the best-first search paradigm to obtain the candidate set. Specifically, we maintain a priority queue PQ with entries in the form of (*mdist, cellID, q*), where *mdist* is the minimum distance from the cell c with *cellID* to the query point $q \in Q$. *mdist* is used as the key to sort the entries in PQ. The retrieval process starts from inserting all cells in the highest level of HICL that contains any query activity to PQ. Then similar with the best-first search, the algorithm repeatedly dequeues the top entry of PQ, i.e., the one having the smallest *mdist* with respect to some query point q_i. If it is not the leaf cell (the cell in the lowest level of HICL), inserts its child cells that contains any activity in $q_i.\Phi$ back to PQ. This can be done by looking up the HICL for each $\alpha \in q.\Phi$ and take the union set of the cells in the inverted list. By this means, the cells without any query activity will be pruned

automatically. Once the popped out entry refers to a leaf cell, the algorithm checks the ITL (if exists) of this cell for each activity in q and put the trajectories in the ITL into a candidate set CS. This procedure continues until CS contains at least λ candidates.

B. Computing Lower Bound Distance

Another important task during the candidate retrieval process is to maintain and update a lower bounding distance D_{lb} for all "unseen" trajectories. A straightforward approach is to directly use the *mdist* in the top entry of PQ, which obviously is a lower bound for D_{mm}. However it is too loose to be useful in practice. To develop a tighter lower bound, we use a set of sorted lists, $cells_n(q_i)$, to keep track of the IDs of the m nearest cells with respect to q_i, which have not been visited yet. At first, all the lists are initialized empty. Then whenever an entry (*mdist, cellID,* q_i) is dequeued from PQ, we first remove *cellID* from $cells_n(q_i)$ (if it exists), and then insert the IDs of its child cells, which contains any activity of $q_i.\Phi$, into $cells_n(q_i)$. The cells in $cells_n(q_i)$ are sorted ascendingly based on their *mdist* to q_i, and only the first m cells are kept. Now we propose to use these cells to derive a tighter lower bounding distance, shown in Algorithm 2.

For the convenience of presentation, we only describe how the lower bound D^i_{lb} with respect to each q_i is derived, and D_{lb} is just the sum of D^i_{lb} for all $q_i \in Q$. First, for each cell c_j in $cells_n(q_i)$, we create a virtual point p_j with all the activities in c_j, which can be acquired directly from ITL of the index, and $d(p_j, q) = mdist(c_j, q)$ (line 6). Then we use these points to create a trajectory $Tr_i = (p_1, p_2, ..., p_m)$ and compute the minimum point match distance $D_{mpm}(q_i, Tr_i)$ by using the Algorithm 3 (line 7). Finally the smaller value between $D_{mpm}(q_i, Tr_i)$ and $d(q_i, c_m)$ is chosen as the lower bound of D_{mpm} for all "unseen" trajectories (line 8).

Algorithm 2: Lower Bound Construction

```
   Input: Q, cells_n(q_i) for q_i ∈ Q
   Output: D_lb
1  mdist ← minimum distance in the top entry of PQ;
2  D_lb ← 0;
3  for each (q_i, Φ_i) ∈ Q do
4      if cells_n(q_i) is not empty then
5          for each cell c_j in cells_n(q_i) do
6              Create a point p_j s.t. p_j.Φ = c_j.Φ and
               d(q_i, p_j) = mdist(q_i, c_j);
7          Create a trajectory Tr_i = (p_1, p_2, ..., p_m);
8          D_lb ← D_lb + min(D_mpm(q_i, Tr_i), d(q_i, p_m));
9      else
10         D_lb ← D_lb + mdist;
11 return D_lb;
```

Theorem 1: D_{lb} derived by Algorithm 2 lower bounds D_{mm} for all "unseen" trajectories in the database.

Proof: Since D_{mpm} is the minimum match distance we can get given the virtual trajectory points and each virtual point is the optimal one amongst all the points in the cell, D^i_{lb} is guaranteed to lower bound the minimum point match distance between q_i and all unvisited trajectories. Consequently, D_{lb} lower bounds the minimum match distance between Q and all unvisited trajectories. ■

C. Validating Candidates

This step validates each candidate trajectory whether it contains all the activities specified by the query. Recall that we have built a data structure, called *trajectory activity sketch (TAS)*, as a component of the index, which will be used to prune a number of candidates without probing the original trajectories on the disk. Given a candidate $Tr \in CS$, we firstly check if its *TAS* enclose all the activities in the query, i.e., $\forall \alpha \in Q.\Phi, \alpha.ID \in TAS(Tr)$. Clearly, this may introduce false positives, i.e., the *TAS* covers the query activities but the trajectory does not actually. However it guarantees that no false dismissals will be introduced.

Consider the example in Figure 2. If Tr_3 is retrieved as a candidate, we can check that its activity sketch $[b, c] \cup [e, f]$ does not contain the query activities a and d. Hence Tr_3 is not a valid candidate.

To further eliminate the false positives, we then obtain the *activity posting list (APL)* for each remaining candidate to see if there exists a posting list for each query activity. If yes, then the trajectory is set to be valid; otherwise, it will be removed from the candidate set.

D. Computing Minimum Match Distance

The last step is to compute the minimum match distance with respect to Q for each valid candidate. According to Lemma 1, it is equivalent to find the minimum point match for each query point in Q. For the sake of simplicity, we just describe the algorithm of computing the minimum point match for a single query point in the sequel.

A straightforward way to evaluate the minimum match distance is to enumerate all match point set in the candidate Tr and find the one with the smallest distance with the query q. Obviously this will involve a large number of combinations, which makes the computation cost too high. We propose a smarter and more efficient algorithm to compute the minimum match distance, which can reduce the number of combinations and terminate early.

The basic structure is illustrated in Algorithm 3. Given a query point $q \in Q$ and a valid candidate Tr, we firstly retrieve the points in the *activity posting list* (APL) of Tr for each $\alpha \in q.\Phi$ into a candidate point set, denoted as CP. The points in CP are then sorted according to their distances with respect to q, the purpose of which is to terminate the algorithm early. We also maintain a hash table $\mathcal{H}$ whose key is a subset of the query keyword set and its value is the current minimum match distance with respect to this keyword subset. Next the algorithm sequentially processes each point p in CP. For each point p, we only care about the set of keywords that overlap with the query, denoted as $p.\Phi'$, and push it into an FIFO queue $\mathcal{L}$. After that the algorithm iteratively pops out the first entry of $\mathcal{L}$ (it is $p.\Phi'$ at the beginning). If a better minimum

match for the keyword subset ks already exists in $\mathcal{H}$, then no update is needed. Besides, there is also no need to examine the subsets of ks since there must be a better minimum match for the subset as well. Otherwise, we update the current best minimum match distance for ks to be the distance between p and q, and at the same time put all the $(|ks|-1)$-size subsets of ks into $\mathcal{L}$. Since the minimum match distance for keyword set ks has changed, the minimum match distance for the superset of ks may be affected as well. To ensure the minimum match distances for all existing keyword sets up-to-date, we retrieve each keyword set s from $\mathcal{H}$, generate a new keyword set key which is the union of s and ks, and update the minimum match distance for key if necessary. Note that the keyword set that is a subset or superset of ks can be skipped safely, since in either case, i.e., $key = s$ or $key = ks$, we have $\mathcal{H}[key] < \mathcal{H}[s] + \mathcal{H}[ks]$.

The early termination condition of Algorithm 3 lies in the beginning of each iteration of the candidate points CP (line 5). The algorithm can terminate if the distance between q and the next unchecked point of CP is greater than D_{mpm}. This ensures that all the unchecked points in CP cannot lead to a better minimum match distance since the distance between q_i and any of those single points already exceeds the current D_{mpm}.

Algorithm 3: Minimum Point Match Distance

Input: query point q, candidate Tr
Output: $D_{mpm}(q, Tr)$

1 $CP \leftarrow$ the points in $Tr.APL$ for $\alpha \in q.\Phi$;
2 Sort the points of CP by the distance with q;
3 Initialize a hash table $\mathcal{H}$ to store the current minimum match distance for each subset of query keywords;
4 **for** *each point* $p \in CP$ **do**
5 **if** $\mathcal{H}.hasKey(q.\Phi)$ *and* $\mathcal{H}[q.\Phi] \leq d(p,q)$ **then**
6 ***Break***;
7 $p.\Phi' \leftarrow p.\Phi \cap q.\Phi$;
8 Initialize an FIFO queue $\mathcal{L}$ to store the subsets of $p.\Phi'$;
9 $\mathcal{L}.push(p.\Phi')$;
10 **while** $ks \leftarrow \mathcal{L}.pop()$ **do**
11 **if** $\mathcal{H}.hasKey(ks)$ *and* $\mathcal{H}[ks] \leq d(p,q)$ **then**
12 ***Continue***;
13 **else**
14 $\mathcal{H}[ks] \leftarrow d(p,q)$;
15 Push all the $(|ks|-1)$-size subsets of ks into $\mathcal{L}$;
16 **for** *each* $s \in \mathcal{H}.keys$ **do**
17 **if** $ks \not\subseteq s$ *and* $s \not\subseteq ks$ **then**
18 $key \leftarrow ks \cup s$;
19 $\mathcal{H}[key] \leftarrow \min\{\mathcal{H}[key], \mathcal{H}[s] + \mathcal{H}[ks]\}$;
20 **return** $D_{mpm}(q, Tr) = \mathcal{H}[q.\Phi]$;

Now we use the following example to illustrate how D_{mpm} is computed, where the query point q has the activity set $\{a, b, c, d\}$. We assume all the points in CP are already sorted according to their distances to q. The algorithm will process each point sequentially and update $\mathcal{H}$, D_{mpm} when necessary. The intermediate status of the hash table $\mathcal{H}$ is shown in the right column of Table II.

TABLE II: Example of D_{mpm} computation

CP	$d(p,q)$	Updates of $\mathcal{H}$	D_{mpm}
$p_1 : \{a\}$	10	$\{a\} : 10$	
$p_2 : \{b, c\}$	11	$\{b\} : 11$, $\{c\} : 11$, $\{b, c\} : 11$, $\{a, b\} : 21$, $\{a, c\} : 21$, $\{a, b, c\} : 21$	
$p_3 : \{a, b\}$	13	$\{a, b\} : 13$	
$p_4 : \{d\}$	15	$\{d\} : 15$, $\{a, d\} : 25$, $\{b, d\} : 26$, $\{c, d\} : 26$, $\{b, c, d\} : 26$, $\{a, b, d\} : 28$, $\{a, c, d\} : 36$, $\{a, b, c, d\} : 36$	36
$p_5 : \{c, d\}$	17	$\{c, d\} : 17$, $\{a, c, d\} : 27$, $\{a, b, c, d\} : 30$	30
$p_6 : \{a, b, c\}$	26	no update since $\mathcal{H}[\{a, b, c\}] = 21 < 26$	
$p_7 : \{a, b, c, d\}$	31	algorithm can stop now since $D_{mpm} = 30 < 31$	

VI. Order-sensitive Query Extension

The similarity query defined in Section II does not take into account the order of the query points. In other words, as long as a trajectory matches all the query activities, it is regarded as a whole match to the query. Though this definition offers some flexibility, sometimes the user may be more interested in the trajectories whose activity order is the same as the query's. In this section, we extend the ATSQ to be order-sensitive, and develop novel algorithms to address the new challenge brought by this extension.

A. Order-sensitive Similarity Query

Definition 7 (Order-sensitive Match): Given a query Q, we say another trajectory Tr is an order-sensitive match of Q if for each query point $q_i \in Q$, we can find a point match P_i from Tr. Besides, the order of the point matches complies with the respective query points, which means for any pair of query points q_i, q_j $(i < j)$, the index of any point in P_i must be smaller than or equal to the index of any point in P_j. The set of point matches forms the order-sensitive match, denoted by $Tr.OM(Q)$. The order-sensitive match distance is the sum of point match distances, i.e., $D_{om}(Q, Tr.OM(Q)) = \sum_{q_i \in Q} D_{pm}(q_i, P_i)$.

Similarly, we can define the *minimum order-sensitive match* $Tr.MOM(Q)$ to be the one with the smallest D_{om}, which is called the minimum order-sensitive match distance between Q and Tr, denoted as $D_{mom}(Q, Tr)$.

In the running example of Figure 1, the minimum point matches $Tr_1.MPM(q_1) : \{p_{1,2}, p_{1,3}\}$ and $Tr_1.MPM(q_2) : \{p_{1,1}, p_{1,2}\}$ do not comply with the order of q_1, q_2. Hence they cannot constitute the order-sensitive match. Instead, $\{\{p_{1,2}, p_{1,3}\}, \{p_{1,4}, p_{1,5}\}, \{p_{1,5}\}\}$ is an order-sensitive match, and easy to verify that, it is also the minimum order-sensitive match. On the other hand, $Tr_2.MOM(Q)$ is the same as $Tr_2.MM(Q)$.

Order-sensitive Activity Trajectory Similarity Query (OATSQ). Given an activity trajectory set $\mathcal{D}$, a query trajectory Q, a positive integer k, an order-sensitive activity trajectory similarity query (OATSQ) returns k distinct trajectories from $\mathcal{D}$ that have the smallest $D_{mom}(Q, Tr)$.

B. Retrieving and Validating Candidates

It is not hard to observe that the relationship between the minimum point match distance and the minimum match distance stated by Lemma 1 does not hold any more for order-sensitive match distance. This is because the minimum point matches may not comply with the order of the query. In some circumstances, there is no order-sensitive match even though the point match exists for each query point. However, the minimum match distance can serve as a lower bound for the minimum order-sensitive match distance, formally stated as the following lemma.

Lemma 3: Given a query Q and a trajectory Tr, $D_{mm}(Q, Tr)$ always lower bounds $D_{mom}(Q, Tr)$. Besides, this lower bound is tight.

Proof: Since $D_{mm}(Q, Tr)$ is yielded by choosing the minimum point match for each query point, changing to any other point match will result in increase of the overall distance. Besides, when these minimum point matches comply with the order of activities in the query, $D_{mm}(Q, Tr) = D_{mom}(Q, Tr)$, so the lower bound is tight. ■

Based on Lemma 3, we can still adopt the algorithm described in Section V to retrieve the candidates since the lower bound distance for the "unseen" trajectories still applies. After that, the candidates are validated in a similar way as in ATSQ, i.e., checking TAS and then APL of the candidate to see if it contains all the query activities. This is sufficient for ATSQ to ensure all the valid candidates will be the matches with respect to the query definitely. However, for OATSQ a candidate surviving the above check may not be an order-sensitive match to the query due to the additional order constraint. As we shall see in the next subsection, evaluation of D_{mom} is more complex and expensive than D_{mm}, hence further validation is needed to eliminate more invalid candidates.

To this end, for each query point q_i, we look up the APL of the candidate Tr to find out all the points that contain any of the activities in $q_i.\Phi$. Then only the smallest and greatest position indexes of these points are kept, which are called the *matching index bound* of q_i, denoted by $MIB(q_i) = [lb, ub]$. Finally we compare the MIBs of every pair of query points. If there exist two query points $q_i, q_j \in Q$ $(i < j)$ such that $MIB(q_i).lb > MIB(q_j).ub$, then Tr can be eliminated from the candidate set since the point matches for q_i and q_j cannot comply the order of $q_i \rightarrow q_j$.

C. Computing Minimum Order-sensitive Match Distance

Computing D_{mom} is more challenging than D_{mm} since we need to make the order of point matches consistent with the query and try to minimize the match distance at the same time. A straightforward approach is to find out all the point matches for each query point and then try all the possible combinations to find the one with the smallest match distance and satisfying the order constraint. Clearly this approach is not efficient especially when the query trajectory is long and each query point has many point matches. In the sequel, we propose a more efficient solution using dynamic programming.

Given a query $Q : \{q_1, q_2, ..., q_m\}$ and a valid candidate trajectory $Tr : \{p_1, p_2, ..., p_n\}$, we define an $m \times n$ matrix G such that its element $G(i, j)$ $(1 \leq i \leq m, 1 \leq j \leq n)$ represents the D_{mom} between the sub-query $Q[1, i] : \{q_1, ..., q_i\}$ and the sub-trajectory $Tr[1, j] : \{p_1, ..., p_j\}$. Now what we need to do is construct the matrix G until it is filled up, and then $G[m, n]$ holds the desired value of D_{mom} between Q and Tr.

First, we observe the following relationship between a structure and its sub-structures exists in G,

$$G(i, j) = \min_{1 \leq k \leq j} \{G(i-1, k) + D_{mpm}(q_i, Tr[k, j])\} \quad (1)$$

where D_{mpm} is the minimum point match distance between q_i and the sub-trajectory $Tr[k, j]$, which can be computed by the algorithm described in Section V-D.

Intuitively, $G(i, j)$ can be derived by minimizing the D_{mom} between the first $i - 1$ query points and the sub-trajectory $Tr[1, k]$ plus the minimum point match distance between q_i and the sub-trajectory $Tr[k, j]$. It is guaranteed to be an order-sensitive match since q_i is restricted to match the part $Tr[k, j]$ which is successor of $Tr[1, k]$.

Lemma 4: Matrix G has the following two monotonicity properties: 1) If $j' > j$, $G(i, j) \geq G(i, j')$; 2) If $i' > i$, $G(i, j) \leq G(i', j)$.

Proof: 1) $j' > j$ means $Tr[1, j] \subset Tr[1, j']$. So for the same sub-query $Q[1, i]$, it is possible to find a better match that has smaller match distance. 2) $i' > i$ means $Q[1, i] \subset Q[1, i']$. With the same sub-trajectory $Tr[1, j]$, more point match distances are contributed to the overall match distance. Another possibility is that no order-sensitive match exists for $Q[i, j']$. In either case, $G(i', j)$ cannot be smaller than $G(i, j)$. ■

The computation process is illustrated in Algorithm 4. At the beginning we initialize a guardian row of G to be zero for the implementation convenience. Then the algorithm iterates each row and column to fill G progressively. For each pair (i, j), $G(i, j)$ is firstly initialized to be the greatest value. Then with k ranges from j to 1, $G(i, j)$ will be updated once a smaller value is obtained based on Eq. (1). Note that the iterator k is decremented from j, which has two benefits. First, the evaluation of D_{mpm} can be done incrementally since only one more point is added to $Tr[k, j]$ each time. Second, once $G(i-1, k) = +\infty$, we can break the loop since $G(i-1, k')$ is also infinite for any $k' < k$ (based on Lemma 4). Besides, line 9 is an early termination condition, in which after $G(i, |Tr|)$ is derived for each i, we compare it against the current k-th smallest D_{mom} in the result set. If the value already exceeds D^k_{mom}, it is assured that $G(|Q|, |Tr|) > D^k_{mom}$ due to Lemma 4, hence there is no need to continue the computation.

Algorithm 4: Minimum Order-sensitive Match Distance

Input: query Q, candidate Tr, the k-th smallest D_{mom} found so far D^k_{mom}
Output: $D_{mom}(Q, Tr)$

```
G(0, *) ← 0;
for i = 1 to |Q| do
    for j = 1 to |Tr| do
        G(i, j) ← +∞;
        for k = j to 1 do
            if G(i − 1, k) ≠ +∞ then
                Evaluate D_mpm(q_i, Tr[k, j]);
                Update G(i, j) according to Eq.(1);
    if G(i, |Tr|) > D^k_mom then
        return
return G(|Q|, |Tr|);
```

Finally we use the example in Figure 1 to demonstrate how our algorithm works. In order to compute the D_{mom} between Q and Tr_1, we will fill the matrix $G(i, j)$ progressively. First we compute the entries $G(1,1), G(1,2), ..., G(1,5)$, which is equivalent to computing the minimum point match distances between q_1 and sub-trajectories $Tr_1[1:1], Tr_1[1:2], ..., Tr_1[1:5]$ respectively. The results are shown in the first column of Table III. After that we set $i = 2$ to include the second query point q_2 and increases j from 1 to 5. To exemplify this, suppose the next entry to be evaluated is $G(2,3)$. We iterate the variable k from 3 downto 1. When $k = 3$, $G(1,3) + D_{mpm}(q_2, \{p_{1,3}\}) = +\infty$ since $p_{1,3}$ is not a point match of q_2. When $k = 2$, we find $G(1,2) = +\infty$ so it is safe to conclude $G(2,3) = +\infty$ according to Lemma 4. Similarly when $j = 5$, we can get $G(2,5) = G(2,4) + D_{mpm}(q_2, \{p_{1,4}, p_{1,5}\}) = 55$. Table III presents all the entry values after G is filled and $G(3,5)$ holds the value of D_{mom} between Q and Tr_1.

TABLE III: Example of D_{mom} computation

$G(i,j)$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=1$	$+\infty$	$+\infty$	24	24	24
$i=2$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	55
$i=3$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	56

VII. Experiments

We conduct extensive experiments on real trajectory datasets to study the performance of the proposed index and query algorithms.

A. Experimental Settings

We use two real activity trajectory datasets by crawling the online check-in records of Foursquare within the areas of Los Angeles (LA) and New York (NY) [24]. Each check-in record of Foursquare contains the user ID, venue with geo-location (place of interest), time of check-in, and the tips written in plain English. We put the records belonging to the same user in the chronological order to form the trajectory of this user. The activity set for each place of a trajectory is generated by the words/phrases in the tips associated with the location. The detailed statistics of the two datasets are given in Table IV.

TABLE IV: Statistics of datasets

	LA	**NY**
#trajectory	31,557	49,027
#venue	215,614	206,416
#activity	3,164,124	2,056,785
#distinct activity	87,567	64,649

We will compare time cost of the proposed methods (GAT) against the three baseline approaches introduced in Section III, namely inverted list based algorithm (IL), R-tree based algorithm (RT), and IR-tree based algorithm (IRT). Note that the four algorithms only differ in the index structure and how they retrieve candidates, and they will use the same algorithms to compute the minimum match distance (Section V-D) and minimum order-sensitive match distance (Section VI-C).

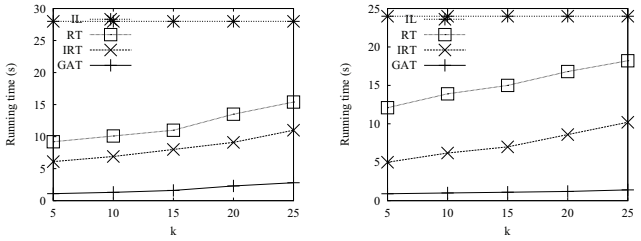
TABLE V: Default parameter values

Parameter	**Default value**
#results k	9
#query points $\|Q\|$	4
#query activities per location $\|q.\Phi\|$	3
the diameter of query $\delta(Q)$	10km

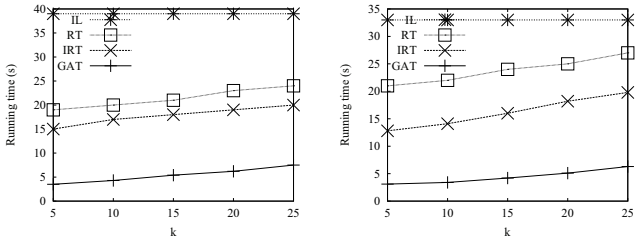
By default, we build a d-Grid with $d = 8$ for the trajectory dataset, which means the entire space is partitioned into $2^8 \times 2^8$ cells. We keep the hierarchical cells from level 1 ($d = 1$) to level 6 ($d = 6$) within main memory, and store the two lowest levels ($d = 7, 8$) on hard disk. The default values for other parameters are summarized in Table V. In the experiments, we will vary these values to investigate the effect of each parameter. For each set of experiment, we generate 50 queries and report the average running time. Each query is generated by randomly choosing a trajectory from the dataset, and then selecting the desired number of locations and activities. All the algorithms including the baselines are implemented in Java and run on a PC with Intel Duo-Core 3GHz CPU and 4GB memory.

B. Performance Evaluation

Effect of k. In the first set of experiments, we study the effect of the intended number of results k by plotting the average time costs of ATSQ and OATSQ on both LA and NY datasets. As shown in Figure 3, our proposed indexing approach, *GAT*, significantly outperforms all other three baseline indexing methods on both datasets. In particular, GAT is at least one order of magnitude faster than IL and 4–5 times faster than RT and IRT. Since IL finds all the trajectory candidates that match the query activities first and then compute the match distance for these candidates, the running time remains constant for all values of k. The other three methods, on the other hand, incur higher cost as k increases. This is expected since the kth smallest match distance becomes greater, which means more candidates need to be retrieved and refined. We



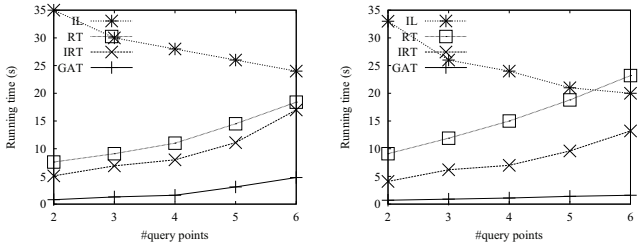
(a) ATSQ on LA (b) ATSQ on NY



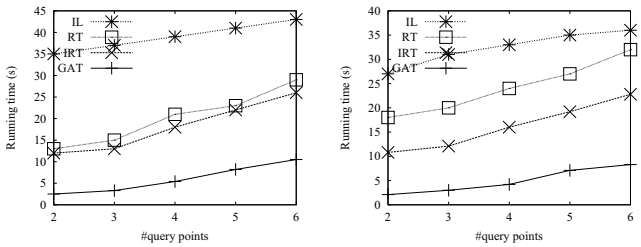
(c) OATSQ on LA (d) OATSQ on NY

Fig. 3: Effect of k

also observe that, though the NY dataset has more trajectories, all algorithms except RT run faster than on LA dataset. This is because trajectories of LA contain more activities averagely, resulting in more candidates matching the query activities. But RT only uses spatial information of trajectories to prune the search space, so it tends to be less effective on larger dataset.



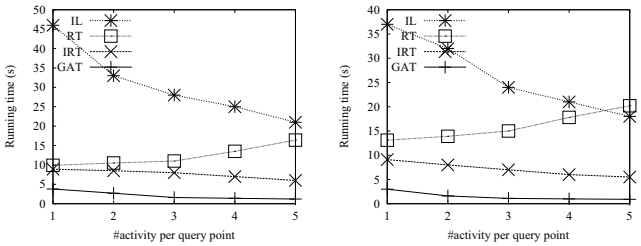
(a) ATSQ on LA (b) ATSQ on NY



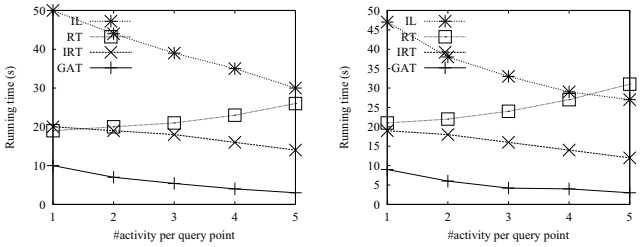
(c) OATSQ on LA (d) OATSQ on NY

Fig. 4: Effect of $|Q|$

Effect of $|Q|$. Next we study the query performance when the number of query locations, $|Q|$, is varying. The results are presented in Figure 4. Again, our proposed method has superior performance than all baseline approaches. With the increase of query points, RT,IRT and GAT incur more time cost since they all utilize spatial index to retrieve candidates around each query point. Hence more query points will result in more candidates retrieved. However, IL behaves differently for ATSQ and OATSQ. IL runs faster for ATSQ when more query points are issued because there are fewer candidates matching all the query activities. Though this is also the case for OATSQ, the runtime cost still increases mainly due to the higher computation cost of D_{omo} with more query locations.



(a) ATSQ on LA (b) ATSQ on NY

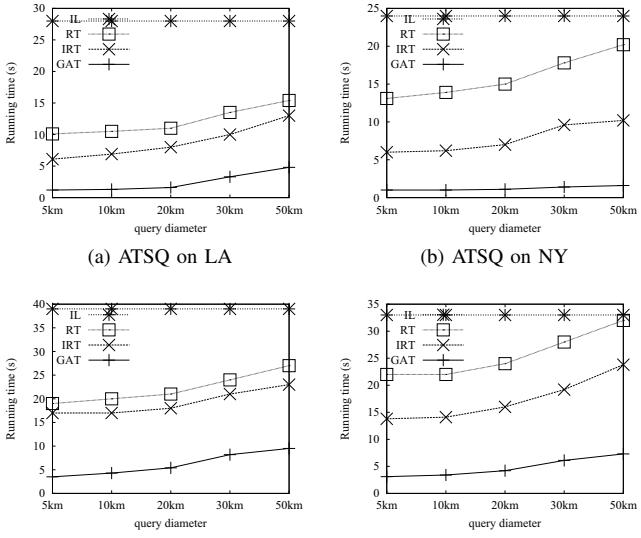


(c) OATSQ on LA (d) OATSQ on NY

Fig. 5: Effect of $|q.\Phi|$

Effect of $|q.\Phi|$. Then we investigate the query performance with regard to the number of activities at each query location, i.e., $|q.\Phi|$. The results are shown in Figure 5. We observe that all approaches except RT consumes less time when $|q.\Phi|$ increases. This is due to the fact that all the three methods make use of activity information when they search for candidates. Therefore more query activities means fewer candidates retrieved and thus less distance computation cost. On the other hand, RT does not incorporate any activity information into the indexing structure, so the candidate retrieval process is not affected by this parameter. However, with more query activities, the k-th smallest D_{mm} or D_{mom} tend to be greater, which means it needs to check more candidates before the result set can be decided.

Effect of $\delta(Q)$. We now proceed to examine the effect of the distribution of query locations. To quantify this factor, we define the diameter of query, $\delta(Q)$, which is the maximum distance amongst all pairs of query points, i.e., $\delta(Q) = \max_{q_i,q_j \in Q} d(q_i, q_j)$. Query with greater diameter is more widely spread over the space. We choose the queries with diameters varying from 5km to 50km and plot the average running time of all approaches in Figure 6. As we can see IL is not affected by this parameter since it does not take into account any geometric property of the query when performing

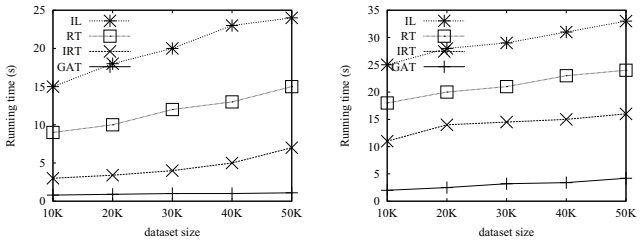


(a) ATSQ on LA (b) ATSQ on NY

(c) OATSQ on LA (d) OATSQ on NY

Fig. 6: Effect of $\delta(Q)$

the search. All other three methods become less efficient when the query points are more distant with each other. This is expected since these methods retrieve the trajectories close to any query point. When the query is more spreaded, more trajectories will be retrieved as candidates.

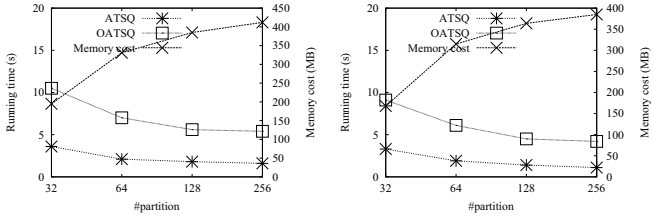


(a) ATSQ (b) OATSQ

Fig. 7: Effect of $|\mathcal{D}|$

Effect of $|\mathcal{D}|$. We also evaluate the scalability of all the approaches. In order to do that, we sample the NY dataset to generate datasets with different number of trajectories varying from 10K to (approx.) 50K, and report the average running time in Figure 7. Without surprise, the time costs of all four methods increase linearly/sublinearly with respect to the size of dataset. But it is worth to note that our proposed method scales much better than the others on both ATSQ and OATSQ.

Effect of partition granularity. Finally we study the effect of the partition granularity of the grid index. Recall that by default we partition the entire space into 256×256 cells ($d = 8$). In this set of experiments, we set the number of partitions to $32 \times 32 (d = 5)$, $64 \times 64 (d = 6)$, $128 \times 128 (d = 7)$ and $256 \times 256 (d = 8)$ and record the respective running time of ATSQ and OATSQ and memory cost of GAT. The results are shown in Figure 8. Generally, better performances will be



(a) LA (b) NY

Fig. 8: Effect of partition granularity

achieved for both ATSQ and OATSQ by using the GAT index with finer granularity since tighter distance lower bound for "unseen" trajectories can be derived with smaller sized cells. But this performance improvement is not so obvious especially when the partition number is beyond 64. This is because, with more partitions, GAT has more levels in the HICL structure, which requires more queue operations (enqueue and dequeue of cells) and neutralizes the benefit gained by the tighter lower bound. The memory cost of the index increases when more cells are built since both HICL and ITL in GAT require more memory. But recall that we only keep the cells in the levels $d > 6$ of HICL on the disk. That means only ITL needs more memory for the cells below this level, which explains the slight increase in memory cost when the partition number is more than 64.

VIII. Related Work

Spatial Keyword Search. Searching spatial objects associated with textual information have gained significant attentions due to the prevalence of spatial web objects on the Internet. The earliest work studying the spatial keyword search problem includes [25][26][27], which retrieve web documents relevant to a keyword query within a pre-specified spatial region. These proposals use loose combinations of an inverted file and a spatial index (e.g., R-tree). The query processing in these proposals occurs in two stages: One type of indexing (e.g. inverted list) is used to filter web document in the first stage, and then the other index (e.g. R-tree) is employed, or the vice versa. This index has the disadvantage that it cannot simultaneously prune the search space using both keywords and spatial distance. More recently, a *location-aware top-k text retrieval* (LkT) query is proposed [22], where the text relevancy to a query is computed by means of language model and a probabilistic ranking function. A new indexing framework, IR-tree that integrates location indexing and text indexing has been developed to efficiently process this query. Variants of LkT query including MkSK query [28] and RSTkNN query [29] have also been proposed. Cao et al. [30] propose a *location-aware top-k prestige-based text retrieval* (LkPT) query, to retrieve the top-k spatial web objects ranked according to both prestige-based text relevance (PR) and location proximity. Zhang et al. [31][32] introduced the m-closest keyword query (mCK query) which aims at finding the closest objects that match the query keywords.

Cao et al. [33] propose a different semantics is taken such that the group of objects in the result covers the query's keywords and has the lowest cost. Yao et al. [34] tackled the problem of answering approximate string match queries in spatial databases. Roy and Chakrabarti [35] studied type-ahead search in spatial databases using materialization techniques. Li et al. [36] studied the problem of direction-aware spatial keyword search, which aims at finding the k nearest neighbors to the query that contain all input keywords and satisfy the direction constraint.

To the best of our knowledge, there is only one work considering the fusion of keywords and trajectories [37]. But in their work, keywords are associated with the whole trajectory rather than each individual point. Hence both the similarity function and the query processing algorithms are quite different from the proposals in our paper.

Trajectory Similarity Search. Due to the structural complexity of trajectory data, measuring the similarity between trajectories is not a straightforward task. Therefore many different similarity functions and algorithms exist to compute the similarity between time series/trajectory data [16][17][18][6][7][19][20]. In particular, the similarity query proposed in [20] is more similar to our work, in which they use multiple locations as the query to search for trajectories that "best match" these locations. However, since they only consider the spatial property of trajectories, their techniques cannot be applied to our problem.

IX. Conclusion

This paper studies the problem of efficient similarity search on the trajectories associated with activity information, given multiple query locations with activity requirement. Two types of queries, ATSQ and OATSQ, are proposed depending on whether the order of query points is considered. To support efficient query processing, we develop a novel hybrid grid index called GAT, and propose efficient algorithms to compute the minimum match distance and minimum order-sensitive match distance between a query and a trajectory. Extensive experimental results based on real datasets demonstrate that the proposed method outperforms several baseline algorithms significantly and achieves good scalability.

Acknowledgement

This work was supported by ARC grants DP120102829 and DP110103423.

References

[1] D. Pfoser, C. Jensen, and Y. Theodoridis, "Novel approaches to the indexing of moving object trajectories," in *VLDB*, 2000, pp. 395–406.
[2] Y. Cai and R. Ng, "Indexing spatio-temporal trajectories with chebyshev polynomials," in *SIGMOD*, 2004, pp. 599–610.
[3] J. Ni and C. Ravishankar, "Indexing spatio-temporal trajectories with efficient polynomial approximations," *TKDE*, vol. 19, no. 5, pp. 663–678, 2007.
[4] V. Chakka, A. Everspaugh, and J. Patel, "Indexing large trajectory data sets with seti," in *CIDR*, 2003.
[5] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in *ICDE*, 2010, pp. 109–120.
[6] M. Vlachos, D. Gunopoulos, and G. Kollios, "Discovering similar multidimensional trajectories," in *ICDE*, 2002, p. 0673.
[7] L. Chen, M. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *SIGMOD*, 2005, pp. 491–502.
[8] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Nearest neighbor search on moving object trajectories," *SSTD*, pp. 328–345, 2005.
[9] K. Zheng, G. Trajcevski, X. Zhou, and P. Scheuermann, "Probabilistic range queries for uncertain trajectories on road networks," in *EDBT*, 2011, pp. 283–294.
[10] K. Zheng, Y. Zheng, X. Xie, and X. Zhou, "Reducing uncertainty of low-sampling-rate trajectories," in *ICDE*, 2012.
[11] J. Lee, J. Han, and K. Whang, "Trajectory clustering: a partition-and-group framework," in *SIGMOD*, 2007, p. 604.
[12] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen, "Discovery of convoys in trajectory databases," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, 2008.
[13] H. Jeung, H. Shen, and X. Zhou, "Convoy queries in spatio-temporal databases," in *ICDE*, 2008, pp. 1457–1459.
[14] Z. Li, B. Ding, J. Han, and R. Kays, "Swarm: Mining relaxed temporal moving object clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 723–734, 2010.
[15] K. Zheng, Y. Zheng, N. J. Yuan, and S. Shang, "On discovery of gathering patterns from trajectories," in *ICDE*, 2013.
[16] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity search in sequence databases," *FDOA*, pp. 69–84, 1993.
[17] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," *ACM SIGMOD Record*, vol. 23, no. 2, pp. 419–429, 1994.
[18] B. Yi, H. Jagadish, and C. Faloutsos, "Efficient retrieval of similar time sequences under time warping," in *ICDE*, 2002, pp. 201–208.
[19] L. Chen and R. Ng, "On the marriage of lp-norms and edit distance," in *VLDB*, 2004, pp. 792–803.
[20] Z. Chen, H. Shen, X. Zhou, Y. Zheng, and X. Xie, "Searching trajectories by locations: an efficiency study," in *SIGMOD*, 2010, pp. 255–266.
[21] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *ACM Sigmod Record*, vol. 14, no. 2, pp. 47–57, 1984.
[22] G. Cong, C. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 337–348, 2009.
[23] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, pp. 1–56, 2006.
[24] J. Bao, Y. Zheng, and M. Mokbel, "Location-based and preference-aware recommendation using sparse geo-social networking data," in *ACM GIS*, 2012.
[25] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Ma, "Hybrid index structures for location-based web search," in *CIKM*, 2005, pp. 155–162.
[26] R. Hariharan, B. Hore, C. Li, and S. Mehrotra, "Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems," in *SSDBM*, 2007, pp. 16–16.
[27] I. De Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008.
[28] D. Wu, M. Yiu, C. Jensen, and G. Cong, "Efficient continuously moving top-k spatial keyword query processing," in *ICDE*, 2011.
[29] J. Lu, Y. Lu, and G. Cong, "Reverse spatial and textual k nearest neighbor search," in *SIGMOD*, 2011.
[30] X. Cao, G. Cong, and C. Jensen, "Retrieving top-k prestige-based relevant spatial web objects," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 373–384, 2010.
[31] D. Zhang, Y. Chee, A. Mondal, A. Tung, and M. Kitsuregawa, "Keyword search in spatial databases: Towards searching by document," in *ICDE*, 2009, pp. 688–699.
[32] D. Zhang, B. Ooi, and A. Tung, "Locating mapped resources in web 2.0," in *ICDE*, 2010, pp. 521–532.
[33] X. Cao, G. Cong, C. Jensen, and B. Ooi, "Collective spatial keyword querying," in *SIGMOD*, 2011.
[34] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou, "Approximate string search in spatial databases," in *ICDE*, 2010, pp. 545–556.
[35] S. Roy and K. Chakrabarti, "Location-aware type ahead search on spatial databases: semantics and efficiency," in *SIGMOD*, 2011, pp. 361–372.
[36] G. Li, J. Feng, and J. Xu, "Desks: Direction-aware spatial keyword search," in *ICDE*, 2012.
[37] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis, "User oriented trajectory search for trip recommendation," in *EDBT*, 2012.