Noname manuscript No.
(will be inserted by the editor)

Snapshot and Continuous Points-based Trajectory Search

**Shuyao Qi · Dimitris Sacharidis ·
Panagiotis Bouros · Nikos Mamoulis**

Received: date / Accepted: date

Abstract Trajectory data capture the traveling history of moving objects such as people or vehicles. With the proliferation of GPS and tracking technologies, huge volumes of trajectories are rapidly generated and collected. Under this, applications such as route recommendation and traveling behavior mining call for efficient trajectory retrieval. In this paper, we first focus on distance-to-points trajectory search; given a collection of trajectories and a set query points, the goal is to retrieve the top-k trajectories that pass as close as possible to all query points. We advance the state-of-the-art by combining existing approaches to a hybrid nearest neighbor-based method while also proposing an alternative, more efficient spatial range-based approach. Second, we investigate the continuous counterpart of distance-to-points trajectory search where the query is long-standing and the set of returned trajectories needs to be maintained whenever updates occur to the query and/or the data. Third, we propose and study two practical variants of distance-to-points trajectory search, which take into account the temporal characteristics of the searched trajectories. Through an extensive experimental analysis with real trajectory data, we show that our range-based approach outperforms previous methods by at least one order of magnitude for the snapshot and up to several times for the continuous version of the queries.

S. Qi, N. Mamoulis
Department of Computer Science
The University of Hong Kong
E-mail: qisy@connect.hku.hk, nikos@cs.hku.hk

D. Sacharidis
Faculty of Informatics
Technische Universität Wien, Austria
E-mail: dimitris@ec.tuwien.ac.at

P. Bouros
Department of Computer Science
Aarhus University, Denmark
E-mail: pbour@cs.au.dk

Keywords Trajectory search · Continuous queries · Spatial proximity

1 Introduction

With the rapid advances in location sensing technologies on one hand, and the widespread adoption of social networking services on the other, the amount of data capturing people's interaction between the physical and digital world is constantly increasing. People nowadays continuously leave behind digital trails of their activities, either *explicitly* by recording their location over time, e.g., their hike, run, ride, or *implicitly* by publishing geotagged content, e.g., tweets, check-ins, photo uploads. In their most primitive form, such trails can be viewed as trajectories, i.e., sequences of spatiotemporal points.

Searching and filtering a large collection of trajectories finds several applications, including route recommendation, behavior mining, and in transportation systems [23, 27]. Different from conventional *shape-based* retrieval task that identifies trajectories similar to a given, or *range-based* retrieval that selects those that cross a specific spatial region, in this paper we focus on *points-based* search, which retrieves trajectories based on given points. In particular, taking as input a set of query points Q (e.g., a particular set of POIs), the *distance-to-points trajectory search* studied in [2, 18] retrieves the trajectories that pass as close as possible to all query points. The distance of a trajectory t to Q is computed by summing up, for each query point $q \in Q$, its distance to the nearest point in t.

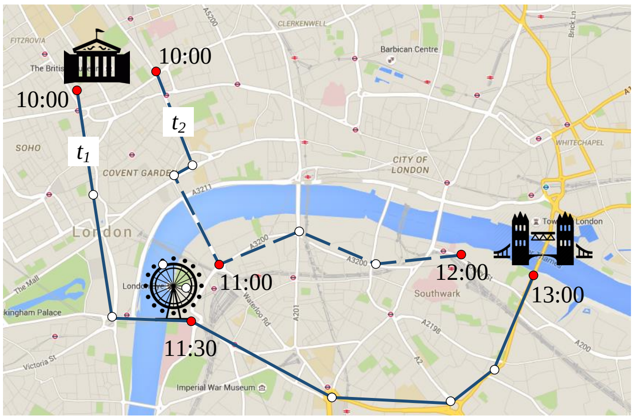


Fig. 1 Example of two touristic trajectories in London.

As a motivating example, consider the collection of touristic trajectories in the city center of London in Figure 1. A travel agency issues a distance-to-points query to survey or recommend routes that pass close to three popular sightseeing attractions, the British Museum, the London Eye and the Tower Bridge; under this task, trajectory t_1 is preferred over t_2. As another example, query set Q could contain traffic congestion points; in this case, the traffic department seeks to discover the causes of the congestion by analyzing the trajectories that pass near the points in Q. In the context of surveillance and security applications, Q may contain locations of crime scenes, and hence the police department issues a distance-to-points query to investigate the correlation of these crime locations by identifying suspects who moved close to all of them.

Contributions. This paper contributes to points-based trajectory search in three directions. First, we thoroughly study the efficient evaluation of *distance-to-points trajectory search*. We review in detail existing algorithms `IKNN` [2] and `GH/QE` [18]. These methods follow a *candidate generation* and *refinement* paradigm, and invoke a nearest neighbor (NN) search centered at each query point to examine the trajectories in ascending order of their distance to Q. By analyzing the pros and cons of these methods, we design a hybrid NN-based algorithm which consistently outperforms `IKNN` and `GH/QE` by over an order of magnitude. Going one step further, we tackle the inherent shortcomings of the NN-based approach itself, namely (i) the increased I/O cost due to independently running multiple NN searches and (ii) the increased CPU cost for continuously maintaining a priority queue for each NN search. We propose a novel *spatial range-based* approach, which is up to 2 times faster than our hybrid algorithm.

Previous work in [2, 18] approached points-based trajectory search as an one-time or a *snapshot* query. As our second line of work, we introduce *continuous* distance-to-points trajectory search, where the query is long-standing and the result set must be maintained whenever updates to the query parameters and/or the data occur. Such a task finds application in data exploration scenarios for example, where the user is not fully aware in advance of the involved query parameters (e.g., the POIs); instead, she actually explores the trajectories in a interactive manner and the answer to one query leads to the formulation of the next. A straightforward solution to continuous distance-to-points trajectory search is to issue a new snapshot query for each occurred update, and then compute the new results from scratch. Naturally, such a solution is expected to perform poorly in practice. Instead, we introduce a continuous counterpart to all NN-based and spatial range-based methods which resume from their previous state and accordingly update the result set. Our tests demonstrated both the efficiency of this incremental approach and the superiority of the continuous spatial range-based evaluation, similar to case of the snapshot distance-to-points trajectory search.

Finally, we observe that the distance-to-points search ranks trajectories solely on how close they pass to the query points in Q, ignoring however other qualitative characteristics of the retrieved results. To fill this gap, we introduce a practical variant that also takes into account the temporal aspect of the trajectories. Specifically, the *span-bounded distance-to-points search* filters out non-interesting trajectories, whose points closest to Q span a time interval greater than a user-defined threshold. Returning to the example of Figure 1,

t_2 is now preferred over t_1 because the travel agency is interested in trajectories that take at most 2 hours to approach the given points of interest. In addition, we also investigate another variant named *span & distance-to-points search*, where the trajectories are ranked both on their distance to the query points and the time interval they span. For all variants, we also consider their *order-aware* counterparts, where Q is a sequence of query points instead of a set, as well as their continuous query counterparts. We note that the snapshot versions of all points-based trajectory search variants were first presented in a previous work of ours [15].

Outline. The rest of the paper is organized as follows. Section 2 formally defines the distance-to-points trajectory search and presents our NN-based and spatial range-based methods. Then, Section 3 investigates the efficient evaluation of the continuous distance-to-points trajectory search. Section 4 formally introduces and addresses the span-bounded distance-to-points and the span & distance-to-points search variants, as well as order-aware counterparts. Section 5 presents our experimental analysis. Finally, Section 6 outlines related work, while Section 7 concludes the paper.

2 Distance-to-Points Trajectory Search

We first investigate trajectory search based on the distance to a set of query points. Section 2.1 formally defines this *distance-to-points search* while Section 2.2 revisits existing work for its evaluation. Next, Sections 2.3 and 2.4 present our novel NN-based and spatial range-based evaluation methods, respectively.

2.1 Problem Definition

Let T be a collection of trajectories. A trajectory in T is defined as a sequence of spatio-temporal points $\{p_1, \ldots, p_n\}$, each represented by a $\langle latitude, longitude, timestamp\rangle$ triple. The input of *distance-to-points trajectory search* over collection T is a set of m spatial query points $Q = \{q_1, \ldots, q_m\}$. Given a query point $q_j \in Q$ and a trajectory $t_i \in T$, we define the $\langle p^*_{ij}, q_j\rangle$ *matching pair* based on the nearest to q_j point p^*_{ij} of trajectory t_i, i.e., $p^*_{ij} = \arg\min_{p \in t_i} dist(p, q_j)$, where $dist(\cdot, \cdot)$ denotes the distance (e.g., Euclidean) between two points in space. We then define the *distance* of a trajectory to Q based on the matching pairs for every query point q_j as:

$$dist(t_i, Q) = \sum_{q_j \in Q} dist(p^*_{ij}, q_j) \quad (1)$$

Consider the example in Figure 2(a), where query points are represented as diamonds, and trajectory points as circles; filled circles indicate matched points of the trajectory to query points. For trajectory t_1, point p^*_{11} is its

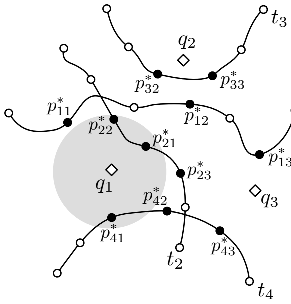


Fig. 2 Distance-to-points trajectory search with 4 trajectories, $T = \{t_1, \ldots, t_4\}$, and 3 query points, $Q = \{q_1, \ldots, q_3\}$; t_1, t_2 is the result to the 2-DTS(T, Q) query.

closest point to query point q_1, and hence $\langle p^*_{11}, q_1 \rangle$ represents a matching pair. The other matched trajectory points of t_1 are p^*_{12} and p^*_{13}. Note that it is possible for a trajectory point to be matched with multiple query points. This is the case with trajectory t_3, where p^*_{32} is the closest point to both q_1 and q_2, i.e., $p^*_{31} \equiv p^*_{32}$.

We now formally define the *distance-to-points trajectory search* problem [2, 18].

Problem 1 (Distance-to-Points Trajectory Search) *Given a collection of trajectories T and a set of query points Q, the k-*Distance-to-Points Trajectory Search*, denoted by k-*DTS(T, Q)*, retrieves a subset of k trajectories $R \subseteq T$ such that for each $t \in R$ and $t' \in T \smallsetminus R$, $dist(t, Q) \leq dist(t', Q)$ holds.*

Returning to the example of Figure 2(a), trajectory t_1 has the lowest distance to Q, followed by t_2, t_3 and t_4; hence, the result to 2-DTS(T, Q) contains trajectories t_1, t_2.

2.2 Existing Methods

Methods `IKNN` [2] and `GH/QE` [18] have previously tackled distance-to-points trajectory search. Note that in [2] the problem was defined with respect to the similarity of a trajectory t_i to the set of query points Q, defined as $sim(t_i, Q) = \sum_{q_j \in Q} e^{-dist(p^*_{ij}, q_j)}$. In what follows, we describe the straightforward adaptation of the `IKNN` algorithm for the distance metric of Equation (1) (which was also used in [18]). The adaptation of `GH/QE` and our methods (Sections 2.3 and 2.4) to the similarity metric of [2] is also straightforward and therefore, omitted. Moreover, the relative performance of all methods is identical and independent of the metric used.

All existing methods adopt a *candidate generation* and *refinement* evaluation paradigm. During the first phase, a set of candidate trajectories C is determined by incrementally retrieving the nearest trajectory points to the query points in Q. For this purpose, the methods utilize a single R-tree to index all trajectory points. A candidate trajectory t is called a *full match* if

Algorithm 1: `IKNN`

Input : collection of trajectories T, set of query points Q, number of results k
Output : result set R
Variables : candidate set C, k-th distance upper bound UB_k, distance lower bound LB

1 initialize $C \leftarrow \emptyset$, $UB_k \leftarrow \infty$ and $LB \leftarrow 0$;
2 **while** $UB_k > LB$ **do**
3   **for** each $q_j \in Q$ **do**
4   δ_j-NN(q_j) $\leftarrow$ the next δ_j nearest trajectory points to q_j;
5   update C with δ_j-NN(q_j);
6   update UB_k and LB; ▷ `Equations (2) and (3)`
7 $R \leftarrow$ `Refine`$_{\mathrm{DTS}}(k, T, Q, C)$;
8 **return** R;

the matching pairs of t to all query points in Q have been identified; otherwise, t is a *partial* match. As soon as the candidate set is guaranteed to include the final results (even as partial matches), candidate generation is terminated, and the refinement phase is then employed to identify and output the results.

In what follows, we detail the candidate generation phase for each method and then, briefly discuss the shared refinement phase.

The `IKNN` Algorithm. Note that the `IKNN` algorithm comes in two flavors; in the following, we consider the one based on best-first nearest neighbor search [7], as it was shown in [2] to be both faster and require fewer I/O operations. Algorithm 1 shows the pseudocode of `IKNN`. During candidate generation (Lines 2–6), the algorithm iterates over the points of Q in a round robin manner. For each query point q_j, the (next) batch of nearest to q_j trajectory points is retrieved using the R-tree index, in Line 4. The nearest neighbor search retrieves a different number of trajectory points δ_j per query point q_j, in order to expedite the termination of this first phase (details in [2]). Based on the newly identified matching pairs that involve q_j, the set of candidates C is then updated in Line 5 by either adding new partial matches or filling an empty slot for existing. For each partial match t_i in C, `IKNN` computes an *upper bound* of its distance to Q by setting the distance of t_i to every unmatched query point equal to the diameter of the space (maximum possible distance between two points): [1]

$$\overline{dist}(t_i, Q) = \sum_{q_j \in Q_i} dist(p^*_{ij}, q_j) + |Q \setminus Q_i| \cdot DIAM, \tag{2}$$

where set $Q_i \subseteq Q$ contains all the query points already matched to a point in trajectory t_i. We denote by UB_k the k-th smallest among the distance bounds for the trajectories in C. In addition, `IKNN` computes a *lower bound* LB of the distance to Q for all unseen trajectories (i.e., those not contained in C), by aggregating the distance of the farthest (retrieved so far) trajectory point to

[1] Under the similarity-based definition of DTS in [2], `IKNN` sets empty "slots" to 0.

Algorithm 2: `GH`

Input : collection of trajectories T, set of query points Q, number of results k
Output : result set R
Variables : candidate set C, global heap H

1 initialize $C \leftarrow \emptyset$ and $H \leftarrow \emptyset$;
2 **while** C contains less than k full matches **do**
3   pop $\langle p_{ij}, q_i \rangle$ from H; ▷ `Get the globally nearest trajectory point to some query point`
4   update C with $\langle p_{ij}, q_i \rangle$;
5   push to H the next nearest trajectory point to q_i;
6 $R \leftarrow$ `Refine`$_{\text{DTS}}(k, T, Q, C)$;
7 **return** R;

Algorithm 3: `QE`

Input : collection of trajectories T, set of query points Q, number of results k
Output : result set R
Variables : candidate set C, global heap H, distance lower bound LB

1 initialize $C \leftarrow \emptyset$, $H \leftarrow \emptyset$ and $LB \leftarrow 0$;
2 **while** C contains less than k full matches with $dist(\cdot, Q) \geq LB$ **do**
3   pop $\langle p_{ij}, q_i \rangle$ from H; ▷ `Get the globally nearest trajectory point to some query point`
4   update C with $\langle p_{ij}, q_i \rangle$;
5   push to H the next nearest trajectory point to q_i;
6   complete the most promising partial matches in C; ▷ `Equation (4)`
7   update LB; ▷ `Equation (5)`
8 $R \leftarrow$ `Refine`$_{\text{DTS}}(k, T, Q, C)$;
9 **return** R;

each query point in Q. Formally:

$$LB = \sum_{q_j \in Q} dist(p_j^{\delta}, q_j) \tag{3}$$

where p_j^{δ} is the last trajectory point returned by the NN search centered at q_j.

The candidate generation phase of `IKNN` terminates when $UB_k \leq LB$; in this case, none of the unseen trajectories can have smaller distance to Q compared to the candidates in C. Last, `IKNN` invokes `Refine`$_{\text{DTS}}$ to produce the results.

The `GH`/`QE` Algorithms. Different from `IKNN`, the methods in [18] retrieve trajectory points in ascending order of the distance to their closest query point. Specifically, a *global heap* H is used to retrieve at each iteration the *globally* nearest trajectory point p_{ij} to some query point q_j, and then, to update candidate set C, accordingly. Algorithm 2 shows the pseudocode of `GH`. The candidate generation phase of `GH` terminates as soon as set C contains k full matches (proof of correctness in [18]). Note that these full matches are not necessary among the final results identified in Line 6 during the refinement phase.

In practice, the order imposed by global heap H cannot guarantee a good performance unless both trajectory and query points are uniformly distributed in space. For instance, if a particular query point is very close to many trajectories, GH will generate a large number of partial matches with only that slot filled. Consequently, it will take longer to produce the k full matches needed to terminate the generation phase, and at the same time a large number of candidates would have to be refined. A similar problem occurs when a query point is located away from the trajectories.

To address these issues, Tang et al. [18] proposed an extension to GH termed QE, which periodically fills the empty slots for the partially matched trajectories with the highest potential of becoming results. These are then retrieved from disk, and their actual distance is computed. A trajectory has high potential if it has (i) few empty slots and (ii) small distance in each filled slot with respect to the next point to be retrieved for that slot. These factors are captured respectively by the denominator and enumerator of the following equation:

$$potential(t_i) = \frac{\sum_{q_j \in Q_i} \left(dist(p_j^H, q_j) - dist(p_{ij}^*, q_i)\right)}{|Q \setminus Q_i|} \tag{4}$$

where set $Q_i \subseteq Q$ contains all the query points already matched to a point in t_i, p_j^H is the next nearest trajectory point to q_j contained in heap H and p_{ij}^* is the nearest to q_j point in trajectory t_i.

Algorithm 3 shows the pseudocode of QE. The candidate generation phase of QE terminates when candidate set C contains k full matches (similar to GH), provided however that their distance to Q is smaller than the distance of all unseen trajectories (Line 2) (proof of correctness in [18]). To determine this, QE computes in Line 7, a *lower bound* LB of the distance for the unseen trajectories (similar to IKNN) by aggregating the distance of the next nearest trajectory point to every query point, i.e., the contents of heap H:

$$LB = \sum_{q_j \in Q} dist(p_j^H, q_j) \tag{5}$$

Refinement. Procedure 1 illustrates a high-level pseudocode of the refinement phase employed by all the methods. Briefly, the $\texttt{Refine}_{\text{DTS}}$ procedure examines the trajectories inside the candidates set C in ascending order of a lower bound $\underline{dist}(\cdot, Q)$ on their distance to the query points in Q. For a full match t_i, we have $\underline{dist}(t_i, Q) = dist(t_i, Q)$ while for a partial match the distance of t_i to every unmatched query point q_i is set similarly to computing distance bound LB:

$$\underline{dist}(t_i, Q) = \sum_{q_j \in Q_i} dist(p_{ij}^*, q_j) + \sum_{q_j \notin Q_i} dist(p_j^{lb}, q_j) \tag{6}$$

where set $Q_i \subseteq Q$ contains all the query points already matched to a point in trajectory t_i, and p_j^{lb} is either the last trajectory point returned by the

Procedure 1: `Refine`$_{\mathrm{DTS}}$

Input : number of results k, collection of trajectories T, set of query points Q, candidate set C
Output : result set R
Variables : k-th distance UB_k in R

```
sort C by lower bound dist(·, Q) in ascending order;        ▷ Equation (6)
for each candidate trajectory t_i ∈ C do
    if |R| = k and dist(t_i, Q) ≥ UB_k then
        break;                                              ▷ Result secured
    compute dist(t_i, Q);
    if |R| < k or dist(t_i, Q) < UB_k then
        update R with t_i and update UB_k;
return R;
```

NN search centered at q_j for `IKNN`, i.e., $p_j^{lb} \equiv p_j^{\delta}$ of Equation (3), or the corresponding trajectory point inside heap H for `GH`/`QE`, i.e., $p_j^{lb} \equiv p_j^{H}$ of Equation (5). Each examined trajectory t_i is retrieved from disk and its actual distance $dist(t_i, Q)$ to query set Q is computed. `Refine`$_{\mathrm{DTS}}$ keeps track of the k-th distance to Q computed so far, denoted by UB_k, and terminates as soon as the lower distance bound of current trajectory exceeds or equals UB_k, i.e., when $\underline{dist}(t_i, Q) \geq UB_k$.

2.3 A Hybrid NN-based Approach

The DTS problem can be viewed as a top-k query [3,8]. For each query point q_j, consider a sorted trajectory list T_j, where each trajectory is ranked according to its distance to the query point. Then, the objective is to determine the top-k trajectories that have the highest aggregate score, i.e., distance, among the lists. However, as these lists are not given in advance and constructing them is costly, the goal is to progressively materialize them, until the result is guaranteed to be among the already seen trajectories.

Following the top-k query processing terminology, a *sorted access* on list T_j corresponds to the retrieval of the next nearest trajectory to query point q_j, which in turn may involve multiple trajectory point NN retrievals. In contrast, a *random access* for trajectory t_i on list T_j corresponds to the retrieval of t_i from disk and the computation of its distance to q_j; in practice, once t_i is retrieved, its distance to all query points can be computed at negligible additional cost.

Methods `IKNN`, `GH` and `QE` employ various ideas from top-k query processing (an overview of this field is presented in Section 6). Particularly, `IKNN` performs only sorted accesses and prioritizes them in a manner similar to `Stream–Combine` [5]. Similarly, `GH` performs only sorted accessses but follows an unconventional strategy for prioritizing them, which explains its poor performance on our tests in Section 5. On the other hand, `QE` additionally performs random accesses following a strategy similar to the `CA` algorithm [3] to select which trajectory to retrieve.

In the following, we present a nearest neighbor-based algorithm termed `NNA`, which combines the strengths of `IKNN` and `QE`. In short, it builds upon the `Quick–Combine` top-k algorithm [6] performing both sorted and random accesses to generate the candidate set. `NNA` has the following features. First, similar to `IKNN`, the algorithm retrieves in a round robin manner, batches of nearest trajectory points to each query point in Q. This addresses the weaknesses of `GH` when dealing with non-uniformly distributed data. Second, after performing the nearest neighbor search centered at each query point, `NNA` fills the slots of the trajectories with the highest potential according to Equation (4), similar to `QE`. Finally, `NNA` employs the termination condition of `IKNN` for the candidate generation phase. In practice, `NNA` extends Algorithm 1 by completing the most promising partial matches in C (similar to `QE`), between Lines 5 and 6. Hence, it is able to compute tighter bounds compared to `IKNN` and thus terminate the generation phase earlier. In addition, it produces fewer candidates than `IKNN`, reducing the cost of the refinement phase.

2.4 A Spatial Range-based Approach

We identify two shortcomings of all the NN-based methods previously described. First, each NN search is implemented independently, which means that R-tree nodes and trajectory points may be accessed multiple (up to $|Q|$) times, which increases the total I/O cost. Second, each NN search is associated with a priority queue, whose continuous maintenance increases the total CPU cost.

Our novel *Spatial Range-based* algorithm, denoted by `SRA`, addresses both these shortcomings. Similar to the NN-based approaches, it follows a generation and refinement paradigm. However, to generate the candidate set, it issues a spatial range search of expanding radius centered at each query point in Q. All searches operate on a common set N of R-tree nodes, which avoids accessing nodes more than once and hence saves I/O operations. Moreover, set N needs not be sorted according to any distance, eliminating costly priority queue maintenance tasks. The range-based search for each query point q_j is associated with *current radius* r_j, and is also assigned a *maximum radius* θ_j. As the algorithm progresses, current radius r_j increases while maximum radius θ_j decreases. Candidate generation terminates as soon as $r_j > \theta_j$ for some query point q_j.

Algorithm 4 shows the pseudocode of `SRA`. In Lines 2–4, `SRA` initializes the current and maximum radius for each query point. For the latter, an upper bound UB_k to the k-th smallest distance to Q is computed. In particular, `SRA` invokes a sum-aggregate nearest neighbor (sum-ANN) procedure [13] retrieving trajectory points in ascending order of $\sum_{q_j \in Q} dist(\cdot, q_j)$. Assuming that this procedure retrieves point p_i of trajectory t_i, the sum-aggregate value is an upper bound to the distance of t_i, i.e., $dist(t_i) \leq \sum_{q_j \in Q} dist(p_i, q_j)$. Hence, once points from k distinct trajectories have been retrieved, `SRA` can determine a value for UB_k.

Algorithm 4: `SRA`

Input : collection of trajectories T, set of query points Q, number of results k
Output : top-k list of trajectories R
Variables : candidate set C, k-th distance upper bound UB_k, current r_i and maximum θ_i search radius for each $q_i \in Q$, set of R-tree nodes N

1 initialize $C \leftarrow \emptyset$ and $N \leftarrow$ R-tree root node;
2 compute UB_k invoking a sum-ANN(T, Q);
3 **for** each $q_j \in Q$ **do**
4   initialize $r_j \leftarrow 0$ and $\theta_j \leftarrow UB_k$;
5 **while** $r_j \leq \theta_j$ for all $q_j \in Q$ **do**
6   select current q_c;
7     $r_c \leftarrow r_c + \xi$; ▷ `Increase` r_c `to expand search around` q_c
8   expand from N all nodes that intersect with the disc of radius r_c centered at q_c;
9     $S \leftarrow$ trajectory points within spatial range r_c found during expansion;
10   update C with S;
11   update UB_k; ▷ `Equation (7)`
12   **for** each $q_j \in Q$ **do**
13   update $\theta_j \leftarrow UB_k - \sum_{q_\ell \in Q \setminus \{q_j\}} r_\ell$; ▷ `Reduce maximum radius`
14 $R \leftarrow$ `Refine`$_{\mathrm{DTS}}(k, T, Q, C)$;
15 **return** R;

During the candidate generation phase in Lines 5–13, `SRA` first selects the query point $q_c \in Q$ with the fewest retrieved points so far, and increases its radius by a fixed ξ, so that each location retrieves more or less the same number of points.[2] Then, it extends the range search centered at q_c to new radius r_c. In particular, all nodes in N that intersect with the search frontier are expanded, i.e., replaced by their children (Line 8). During the expansion, all trajectory points within the frontier are collected in set S (Line 9). Upon completion of the expansion, set N contains no R-tree node or point within r_c distance to q_c, or with distance to q_c greater than θ_c, and N will be re-used in further iterations.

After the expansion, `SRA` uses the newly seen trajectory points in S to properly update candidate set C. Note that for each trajectory t_i in C, `SRA` keeps $|Q|$ slots storing the closest trajectory points $t_i.p_j$ seen so far to each query point q_j. A slot is marked *matched* if the corresponding matching pair has been determined, i.e., when $t_i.p_j \equiv p^*_{ij}$. `SRA` in Line 10 performs the following tasks for each point p_x in S; let t_i be the trajectory p_x belongs to. For each slot q_j that is not *matched*, `SRA` checks whether p_x is closer to q_j than $t_i.p_j$, and updates the slot with p_x if true. If the slot for the current query point q_c was among those examined, it is marked as *matched*. The benefits of this update strategy are twofold. First, it guarantees that no matching trajectory point will be missed, even though `SRA` does not access p_x again (removed from N) for $q_j \neq q_c$. At the same time, it also helps to derive a tighter upper bound

[2] In the future, we plan to investigate variable ξ_j values based on current radius r_j and the trajectory point density around q_j, inspired by determining δ_j value in [2].

for the distance of t_i:

$$\overline{dist}(t_i, Q) = \sum_{q_j \in Q_i} dist(p^*_{ij}, q_j) + \sum_{q_j \in Q \setminus Q_i} dist(t_i.p_j, q_j). \tag{7}$$

Compared to Equation (2) utilized by `IKNN` and `NNA`, Equation (7) computes a tighter bound on unmatched slots. Based on these bounds, a tighter value for UB_k can be established (Line 11).

To better explain the procedure in Line 10, we use the example of Figure 2(a) for $k = 2$. `SRA` has just started and thus C is empty. Assume that the current query point is $q_c = q_1$, and let $r_1 = 0+\xi$ be the radius of the shaded disk depicted in the figure. As a result, set S in Line 9 contains trajectory points $\{p^*_{21}, p^*_{22}, p^*_{41}\}$. Moreover, candidate set C contains t_2 and t_4. For trajectory t_2, p^*_{21} is settled as the matching point to q_1 because $dist(p^*_{21}, q_1) < dist(p^*_{22}, q_1)$ and no unseen point of t_2 can be closer. On the other hand, the matching points to q_2, q_3 cannot be yet determined, but we can use p^*_{21} and p^*_{22} to bound t_2's distances to q_2 and q_3. Therefore, the slots for t_2 become $\langle \mathbf{p^*_{21}}, p^*_{22}, p^*_{21} \rangle$, where bold indicates a *matched* slot. Moreover, an upper bound to the distance of t_2 is determined as $\overline{dist}(t_2, Q) = dist(p^*_{21}, q_1) + dist(p^*_{22}, q_2) + dist(p^*_{21}, q_3)$. Similarly, we obtain the slots for t_4 as $\langle \mathbf{p^*_{41}}, p^*_{41}, p^*_{41} \rangle$.

As a last step, `SRA` updates the maximum radius for all query points with respect to the new UB_k in Lines 12–13. Observe that `SRA`'s termination condition for candidate generation is essentially identical to that of `IKNN`. Any trajectory not in the candidate set C must have distance to each q_j at least θ_j, and thus distance at least equal to $LB = \sum_{q_j \in Q} \theta_j$. The termination condition of Line 5, $r_j > \theta_j$ for some q_j, and the update of θ_j, imply that, when candidate generation concludes, $UB_k \leq LB$.

Finally, the performance of `SRA` can be enhanced following the key idea of `QE` to further improve the $\overline{dist}(t_j, Q)$ bound and therefore, UB_k. We denote this extension to the `SRA` algorithm by `SRA`+. Specifically, in between Lines 10 and 11 in Algorithm 4, `SRA`+ fills the empty slots of the trajectories in C with the highest potential as computed using Equation (4).

3 Continuous Distance-to-Points Trajectory Search

The previous section approached distance-to-points trajectory search as a *snapshot* (one-time) query. In this section, we investigate the case when the search is continuous (long-standing) and the result set R must be maintained whenever updates to the query and/or the data occur. In particular, we consider three types of updates: (i) *increase/decrease* the number of results k, (ii) *insert/delete* a trajectory and (iii) *insert/delete* a query point. To efficiently evaluate continuous distance-to-points trajectory search, we introduce a continuous counterpart to each of `IKNN`/`NNA`, `GH`/`QE` and `SRA`/`SRA`+; the idea is to resume a DTS algorithm from its previous state and accordingly update result set R. The execution of all continuous algorithms go over the same three phases which essentially extend the candidate generation and refinement evaluation paradigm.

Phase 1: *State update*. The algorithm updates its current state according to the occurred update.

Phase 2: *Candidate generation*. If necessary, the main loop of candidate generation is resumed to include additional trajectories in candidate set C. The algorithm enters Phase 2 if the termination condition for candidate generation is not met (Line 2 in Algorithms 1, 2, 3 and Line 5 in Algorithm 4).

Phase 3: *Refinement*. If necessary, the algorithm refines candidate set C to update result set R.

Before discussing in detail every continuous algorithm, we briefly outline how the various update types affect the phases to be executed and result set R.

Increase k. With this update, additional trajectory results are requested. A continuous DTS algorithm may enter Phase 2 to retrieve additional candidates until the termination condition of candidate generation is met. In any case, the algorithm enters Phase 3 to further refine candidate set C and produce the updated result set R.

Decrease k. With this update, fewer results are requested, which makes the maintenance of result set R straightforward. Phase 2 is not necessary as the algorithm already has the appropriate number of candidate trajectories, while Phase 3 performs no refinement; the algorithm needs only to truncate the previous result set. This trivial case of update will not be further discussed.

Insert q_{new}. In this case, results with respect to an additional new query location are requested, i.e., q_{new} is added to set Q. Hence, a new slot has to be accounted for, which means that the algorithm may enter Phase 2 to identify additional candidates and Phase 3 to refine the updated candidates set C.

Delete q_{old}. With this update, an existing location is no longer relevant to the DTS query, i.e., q_{old} is removed from set Q. The q_{old} slot disappears, which may force a continuous DTS algorithm to enter Phase 2 to produce additional candidates. In any case, refinement in Phase 3 is required.

Insert t_{new}. In this case, a new trajectory t_{new} is added to collection T. A continuous DTS algorithm treats t_{new} as a full match; i.e., all its matching pairs are immediately identified and t_{new} is inserted to candidate set C. The algorithm does not enter Phase 2 as set C still contains enough candidates, while Phase 3 does not need to refine the candidate trajectories. The updated result R can be obtained by potentially evicting the previous k-th result to include t_{new}.

Delete t_{old}. With this update, an existing trajectory t_{old} is removed from collection T. Note that the case when t_{old} was never part of candidate set C is trivial and hence will not be further discussed. Otherwise, the algorithm enters Phase 2 to retrieve additional candidates and Phase 3 to refine set C only if the t_{old} trajectory was part of the result set R.

3.1 Algorithms IKNN and NNA

The state of the continuous IKNN/NNA algorithms includes the result set R, the candidate set C, the upper bound $\overline{dist}(t_i, Q)$ of the distance to the set Q for all $t_i \in C$, and bounds UB_k and LB.

Increase k. In Phase 1 where the state of the algorithms is updated, only the upper bound UB_k must be updated (due to the new value of k). Then, IKNN/NNA enter Phase 2 (Lines 2–6 of Algorithm 1) to identify additional candidate trajectories only if the updated UB_k is smaller than the unaffected lower bound LB. Nevertheless, refinement must be executed in Phase 3 to identify the additional required results. Note that $\texttt{Refine}_{\text{DTS}}$ does not run from scratch; in practice, the procedure examines only the new candidates to determine whether they can be part of the result set R, based on their $\underline{dist}(\cdot, Q)$, and then, computes $dist(\cdot, Q)$, if needed.

Insert q_{new}. In Phase 1, a new unmatched slot is created for each candidate trajectory. Consequently, the upper distance bound $\overline{dist}(t, Q)$ for each trajectory $t \in C$ is incremented by the $DIAM$ term based on the q_{new} slot. Upper bound UB_k is also incremented by the same term, while lower bound LB remains the same. As a result, IKNN/NNA (almost certainly) enter Phase 2 with the primary goal of filling the slots for q_{new}. Thus, the algorithms invoke a NN search around q_{new} and retrieve roughly as many points were retrieved by the NN search for each of the other slots. At this point, if the termination condition $UB_k \leq LB$ for candidate generation is still not met, the algorithms proceed retrieving trajectory points from all query points in a round robin manner, similar to Algorithm 1. Finally, Phase 3 is always executed to refine the candidate set C and construct result set R.

Delete q_{old}. In Phase 1, the slot for query point q_{old} disappears from each candidate trajectory. Hence, the upper distance bound $\overline{dist}(t, Q)$ of each trajectory $t \in C$ is appropriately decreased and as a result, the upper bound UB_k also decreases. Similarly, the LB lower bound has one term less and thus decreases. However, bound UB_k could decrease less than LB does, which means that the termination condition for candidate generation may not hold and therefore, IKNN/NNA should enter Phase 2. Nevertheless, the algorithms enter Phase 3 for refinement.

Insert t_{new}. In Phase 1, the new trajectory t_{new} is inserted in candidate set C as a full match, i.e., the points on t_{new} are examined to determine the matching pairs for every $q_i \in Q$, and to compute the exact distance $dist(t_{new}, Q)$. Then, IKNN/NNA compare and update (if needed) current upper bound UB_k with $dist(t_{new}, Q)$. As already discussed, Phase 2 is not necessary while the algorithms enter Phase 3 only if UB_k has changed; in that case, $\texttt{Refine}_{\text{DTS}}$

does not run from scratch as the updated result set is immediately constructed by evicting the previous k-th result and inserting t_{new}.

Delete t_{old}. In Phase 1, the algorithms update (specifically increase) upper bound UB_k only if the removed trajectory t_{old} was contained in the result set R. In that case, `IKNN`/`NNA` also enter Phases 2 and 3.

3.2 Algorithms `GH` and `QE`

The state of the continuous `GH`/`QE` algorithms includes the result set R, the candidate set C, the global heap H and in case of `QE`, also the lower bound LB.

Increase k. In Phase 1, the state of the algorithms needs no update as none of R, C, H or LB is directly related to the number of results k. However, as candidate set C now contains less than k full matches, `GH`/`QE` enter Phase 2 (corresponding to Lines 2–5 of Algorithm 2 for `GH` and Lines 2–7 of Algorithm 3 for `QE`), and subsequently, Phase 3 to refine the updated set of candidates.

Insert q_{new}. During Phase 1, a new empty slot is created for each candidate trajectory and consequently, the global heap H is updated in order to anticipate the new NN search centered at q_{new}. Then, `GH`/`QE` enter Phase 2, to perform the NN search for q_{new}, and possibly expand the search around other query points as necessary. Ultimately, refinement in Phase 3 is executed for both algorithms.

Delete q_{old}. In Phase 1, the q_{old} slot for each candidate trajectory in set C is first removed, and then the global heap H is properly updated to reflect this change. Also, in case of `QE`, the lower bound LB is decreased as it has one term fewer.

Regarding candidate generation, `GH` does not enter Phase 2; the termination condition on Line 2 of Algorithm 2 is met since candidate set C still contains k full matches even after removing q_{old} and its slot. On the other hand, `QE` may enter Phase 2 despite already having k full matches as its termination condition on Line 2 of Algorithm 3 additionally requires that these k full matches have a distance to query set Q above LB. In practice, the subtracted term based on the q_{old} slot for each of the k full matches could be either greater (i.e., equal to $DIAM$ if the q_{old} slot was unmatched) or less (if q_{old} was matched) than the corresponding term for LB. Finally, both algorithms enter Phase 3.

Insert t_{new}. In Phase 1, the new trajectory t_{new} is inserted in the candidate set C as a full match. Hence, neither algorithm enters Phase 2 as their termination condition for candidate generation still holds. In contrast, both `GH`/`QE` enter Phase 3 to determine whether t_{new} should evict the previous k-th result.

Delete t_{old}. In Phase 1, the deleted trajectory t_{old} is removed from candidate set C. Then, the algorithms enter Phases 2 and 3 only if t_{old} was one of the k full matches (with distance above LB for the `QE` algorithm).

3.3 Algorithms SRA and SRA+

The state of the continuous SRA/SRA+ algorithms includes the result set R, the candidate set C, the upper bound $\overline{dist}(t_i, Q)$ of the distance to the set Q for all $t_i \in C$, the common set of R-tree nodes N, the current r_i and the maximum θ_i radius for all $t_i \in C$ and last, the upper distance bound UB_k.

Increase k**.** During Phase 1 and the state update of the algorithms, the UB_k bound is re-computed based on the existing candidate trajectories in C; in practice, UB_k is increased (unless there were ties at the k-th place of the previous result set R) and so are the θ_j radii of the query points in Q. This change raises the following critical challenge. Recall from Section 2.4 that SRA/SRA+ have discarded from common set N, R-tree nodes which were outside the maximum radii at the time (Lines 8–9 of Algorithm 4); however, after increasing k, the pruned nodes may contain points from trajectories belonging to the new result set. To guarantee correctness, we need to consider all such R-tree nodes and trajectory points.

For this purpose, the continuous version of SRA/SRA+ employs an additional variable, set N_d which stores the R-tree nodes pruned by the candidate generation on Line 8. Upon increasing the number of results k, the algorithms first update the set of nodes N by merging previous N with N_d and then, update accordingly set S with the trajectories points within distance r_j from each query point $q_j \in Q$, and candidate set C. Finally, the algorithms enter Phase 2 and 3 which correspond to Lines 5–13 and Line 14 of Algorithm 4, respectively.

Insert q_{new}**.** In Phase 1, the current radius r_{new} only for the new query point is set to zero while all previous radii r_j remain the same. Then, for each candidate trajectory in C, an additional slot for q_{new} is created; the slot is marked as *unmatched* and populated with the best trajectory point among those in the other slots. Consequently, the upper distance bound $\overline{dist}(t_i, Q)$ for each $t_i \in C$ is re-computed using Equation (7) with $Q \cup \{q_{new}\}$ and as a result, the UB_k bound is also updated (more precisely, increased due to an additional distance term). Finally, the maximum radius θ_j for all query points $Q \cup \{q_{new}\}$ is updated invoking Lines 12–13 in Algorithm 4.

In practice, the maximum radii θ_j are all increased while the sum over current radii is unaffected. This change results in the same challenge as the case of increasing k; hence, to guarantee correctness the set of R-tree nodes N is accordingly updated using the previously pruned nodes in set N_d. At this point, there is a second challenge to complete Phase 1 and the state update; the trajectories points pruned on Line 9 of Algorithm 4 may provide matching pairs for the new query point q_{new}. For this purpose, we further extend the continuous SRA/SRA+ to store these pruned points in a new set S_d, similar to the case of pruned R-tree nodes. Upon completion of this procedure, the state of the algorithms is properly updated and the algorithms enter Phase 2 and 3.

Delete q_{old}. During Phase 1, both the current and the maximum radius for the q_{old} query point are removed while the corresponding slot is eliminated from all trajectories in C. As a result, their upper distance bound is re-computed; in practice, $\overline{dist}(\cdot, Q)$ decreases and so is the UB_k upper bound, due to removing one distance term. Moreover, maximum radii θ_j are re-calculated. In general, radii θ_j may increase which means that SRA/SRA+ need to update the set of R-tree nodes N using the previously pruned nodes in set N_d (similar to the case of increasing k or inserting a new query point q_{new}) and afterwards, also candidate set C. Subsequently, the algorithms may enter Phase 2 to identify additional candidate trajectories, while Phase 3 is required nevertheless.

Insert t_{new}. In Phase 1, trajectory t_{new} is added to candidate set C and the upper bound UB_k is accordingly updated. In practice, bound UB_k may only reduce and so do the maximum radii. As a result, the termination condition for candidate generation on Line 5 of Algorithm 4 still holds which means that neither of the algorithms enters Phase 2. On the other hand, both SRA/SRA+ enter Phase 3 but only to include t_{new} in result R if $dist(t_{new}, Q) < UB_k$.

Delete t_{old}. In Phase 1, trajectory t_{old} is removed from candidate set C. In addition, if t_{old} was part of the previous result set R, the UB_k upper bound as well as maximum radii need to be re-computed. In this case, SRA/SRA+ both enter Phase 2 and 3.

4 Variants of Points-based Trajectory Search

In this section, we present several extensions to the distance-to-points trajectory search of Sections 2 and 3. Specifically, Sections 4.1 and 4.2 introduce novel points-based trajectory search problems which additionally take into account the temporal aspect of the trajectories, besides their spatial proximity to the query points in Q. Then, Section 4.3 investigates the case of points-based trajectory search when a visiting order is imposed on the query points. Finally, Section 4.4 discusses the continuous counterpart of these queries.

4.1 Span-Bounded Distance-to-Points Trajectory Search

Let P_i^* be the set of all matching pairs for a trajectory t_i, sorted ascending on the timestamp of the involved trajectory points. We define the *span* of trajectory t_i with respect to Q, denoted by $span(t_i, Q)$, as the length of the time interval between the first and the last pair in P_i^*, or equivalently:

$$span(t_i, Q) = \max_{q_x, q_y \in Q} (timestamp(p_{ix}^*) - timestamp(p_{iy}^*)) \tag{8}$$

Intuitively, $span(t_i, Q)$ equals the total time needed to reach as close as possible to all query points in Q, following trajectory t_i.

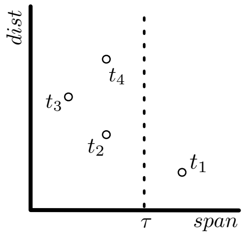


Fig. 3 The *span-dist* plot of trajectories $T = \{t_1, \ldots, t_4\}$ from Figure 2.

Problem 2 (Span-Bounded Distance-to-Points Trajectory Search) *Given a collection of trajectories T, a set of query points Q and a span threshold τ, the k* Span-Bounded Distance-to-Points Trajectory Search*, denoted by k-*BDTS(T, Q, τ)*, retrieves the subset of k trajectories $R \subseteq T$ such that:*

– *for each $t \in R$, $span(t, Q) \leq \tau$ holds, and*
– *for each $t' \in T \setminus R$ with $span(t', Q) \leq \tau$, $dist(t, Q) \leq dist(t', Q)$ holds.*

Consider again the example of Figure 2; for simplicity the trajectory points are reported in fixed time intervals. As a result, the span of a trajectory is proportional to the number of its points from the first to the last matched point (excluding the first). For example, $span(t_1, Q) = 4$ as there are 4 points from p^*_{11} and up to p^*_{13}. Similarly, we obtain the spans of t_2, t_3, t_4 as 2, 1, 2, respectively. Figure 3 plots the trajectories of Figure 2 in the *span-dist* plane. DTS ignores the span values and simply returns the trajectories with the lowest *dist* coordinate. In contrast, BDTS introduces a threshold, e.g., $\tau = 3$, on the span of the trajectories, depicted as the dashed vertical line. Trajectories to the right of this line, i.e., t_1, do not qualify as BDTS results. Therefore, the result of 2-BDTS is t_2, t_3, i.e., the trajectories with the 2 lowest distances among those left of the line. Notice that BDTS may not return the trajectory with the lowest distance to Q if its span exceeds the threshold; e.g., t_1 in Figure 3.

We next discuss the evaluation of the span-bounded distance-to-points trajectory search. Intuitively, k-BDTS(T, Q, τ) is equivalent to a k-DTS(T', Q) distance-to-points query over the subset $T' \subseteq T$ containing trajectories with $span(\cdot, Q) \leq \tau$. However, as $span(t, Q)$ can be computed only after all the matching pairs of a trajectory t to Q are identified, the major challenge is to limit the number of invalid partial matches generated, i.e., those with the $span(\cdot, Q) > \tau$. In the following, we address this issue in two alternative ways.

The idea behind the incremental approach, denoted as `INCREMENTAL`, is to progressively construct the result set R by utilizing the generation phase of a DTS method as a "black" box. Algorithm 5 illustrates `INCREMENTAL`; note that any of the algorithms in Section 2 can be used as the underlying DTS method. At each round, `INCREMENTAL` asks for the missing $k - |R|$ trajectories to complete the result set R in Lines 3–4. For this purpose, a λ-DTS(T, Q) search is processed, with the λ value been increased at each round by $k - |R|$; during

Algorithm 5: `INCREMENTAL`

Input : collection of trajectories T, set of query points Q, span threshold τ, number of results k
Output : result set R
Variables : candidate set C, number of intermediate results λ

1 initialize $C \leftarrow \emptyset$, $R \leftarrow \emptyset$ and $\lambda \leftarrow 0$;
2 **while** $|R| < k$ **do**
3   increase λ by $k - |R|$;
4     $C \leftarrow$ next candidate set of λ-DTS(T, Q);
5     $R \leftarrow R \cup \texttt{Refine}_{\text{BDTS}}(k, T, Q, C, \tau)$;
6 **return** R;

Procedure 2: $\texttt{Refine}_{\text{BDTS}}$

Input : number of results k, collection of trajectories T, set of query points Q, candidate set C, span threshold τ
Output : result set R
Variables : k-th distance UB_k in R

1 sort C by lower bound $\underline{dist}(\cdot, Q)$ in ascending order; ▷ `Equation (6)`
2 **for** each candidate trajectory $t_i \in C$ **do**
3   **if** $|R| = k$ **and** $\underline{dist}(t_i, Q) \geq UB_k$ **then**
4   break; ▷ `Result secured`
5   compute $dist(t_i, Q)$;
6   **if** $span(t_i, Q) \leq \tau$ **then**
7   **if** $|R| < k$ **or** $dist(t_i, Q) < UB_k$ **then**
8   update R with t_i and UB_k;
9 **return** R;

the first round $\lambda = k$. Each time λ is updated in Line 3, the DTS method in Line 4 does not run from scratch. It continues the candidate generation using a new termination condition with respect to the updated λ in order to expand candidate set C. Last, in Line 5, $\texttt{Refine}_{\text{BDTS}}$ illustrated in Procedure 2, examines the new candidates to update result set R by computing their $dist(\cdot, Q)$ and eliminating trajectories with $span(\cdot, Q) > \tau$; in practice, the refinement procedure for BDTS is reminiscent to $\texttt{Refine}_{\text{DTS}}$ of Procedure 1, except for the duration bound check in Line 6.

Intuitively, `INCREMENTAL` takes a conservative approach to BDTS. As it is unable to predict which partial matches could provide a valid trajectory (full match) with $span(\cdot, Q) \leq \tau$, a refinement phase is needed to "clean" the candidate set. Hence, `INCREMENTAL` may involve several rounds of generation and refinement phases. To address these issues, we propose the `ONE–PASS` approach which involves a single generation and refinement round. The idea is again to build upon a DTS method but by extending its candidate generation phase in two ways. First, for each partial match t_i in candidate set C, `ONE–PASS` computes a lower bound of $span(t_i, Q)$ based on the points of t_i matching the current subset of query points $Q_i \subset Q$, as follows:

$$\underline{span}(t_i, Q) = \begin{cases} 0, & \textit{if } |Q_i| = 1 \\ span(t_i, Q_i), & \textit{otherwise} \end{cases} \tag{9}$$

Every partial match with $\underline{span}(\cdot, Q) > \tau$ can be safely pruned. Second, the original termination is triggered only after candidate set C contains at least k valid full matches, i.e., with $span(\cdot, Q) \leq \tau$. This is because the k-th upper bound UB_k of existing candidates can be computed only through full matches. For example, candidate generation of `ONE–PASS` based on `SRA+` terminates as soon as at least k valid full matches are identified and $r_j > \theta_j$ holds for some query point q_j.

4.2 Span & Distance-to-Points Trajectory Search

Despite taking into account their temporal span, the span-bounded distance-to-points trajectory search in Section 4.1 still ranks the trajectories solely on their distance to the query points in Q. Depending on the application, one may consider alternative definitions for points-based trajectory search that take into account both the distance and the span metrics. In this spirit, we consider ranking the results with respect to a linear combination of the *span-dist* metrics:

$$f(t, Q) = \alpha \cdot dist(t, Q) + (1 - \alpha) \cdot span(t, Q) \tag{10}$$

where α weights the importance of each metric. With Equation (10), we introduce the *k Span & Distance-to-Points Trajectory Search*, denoted by k-SDTS(T, Q) which returns the subset of k trajectories $R \subseteq T$ with the lowest $f(\cdot, Q)$ value.

Intuitively, k-SDTS(T, Q) comes as an variation to Problem 1 and the k-DTS(T, Q) query defined in Section 2 by replacing $dist(\cdot, Q)$ with $f(\cdot, Q)$. Consequently, all DTS methods discussed in Section 2 can be extended to evaluate a k-SDTS query. Note that the upper bound $\overline{f}(t, Q)$ of a partial match t can be computed by setting $\overline{span}(t, Q)$ equal to the total duration of the trajectory t. In contrast, as no matching pairs are identified for the unseen trajectories, the lower bound LB or the θ_j values are defined similar to the DTS methods, i.e., essentially setting the lower bound of span to zero.

4.3 Order-aware Points-based Trajectory Search

Similar to [2], we also consider a variation of the points-based trajectory search when a visiting order is imposed for the query points. In this search, the matched trajectory point p^*_{ij} to query point q_j, is not necessarily the nearest to q_j point of trajectory t_i. Consider for example trajectory t_2 in Figure 2. The depicted p^*_{22}, p^*_{21}, p^*_{23} for DTS cannot be the matched points in the $q_1 \rightarrow q_2 \rightarrow q_3$ order-aware DTS, as they violate the visiting order. Instead, the matched points that preserve the imposed visiting order are p^*_{22}, p^*_{22}, p^*_{23}, where p^*_{22} is matched with q_1 although $dist(p^*_{22}, q_1) > dist(p^*_{21}, q_1)$. The distance of a trajectory to sequence Q is recursively defined as follows:

$$dist_o(t,Q) = \begin{cases} \min \begin{cases} dist_o(t,T(Q)) + dist(H(t),H(Q)) - DIAM \\ dist_o(T(t),Q) \end{cases} & \text{if } t \neq \varnothing, Q \neq \varnothing \\ |Q| \cdot DIAM & \text{if } t = \varnothing \\ 0 & \text{if } Q = \varnothing \end{cases} \tag{11}$$

where $H(S)$ is the first point (head) in a sequence S, $T(S)$ indicates the tail of S after removing $H(S)$, $\varnothing$ denotes the empty sequence, and $DIAM$ represents the diameter of the space. The distance can be computed using dynamic programming [2]. To derive an upper bound on a partial matched trajectory t_i, we consider only the subsequence Q_i of Q that contains the matched query points, i.e., $\overline{dist_o}(t_i,Q) = dist_o(t_i,Q_i)$. For order-aware BDTS, distance and its upper bound are the same as in order-aware DTS. Note, however that the lower bound on span (Equation (9)) does not apply as the matching is not yet finalized. For order-aware SDTS evaluation, $f_o(t,Q)$ and its upper bound are defined in a similar manner to order-aware DTS.

4.4 Continuous Points-based Trajectory Search Variants

Similar to the distance-to-points trajectory search in Section 2, we finally discuss how BDTS, SDTS and ordered-aware trajectory search can be approached as continuous queries. To this end, we consider the same types of updates as in Section 3, i.e., increase/decrease k, insert/delete a trajectory and insert/delete a query point. Note also that the methods for the continuous BDTS, SDTS and ordered-aware trajectory search follow the same three-phase evaluation paradigm of state update in Phase 1, candidate generation in Phase 2, and refinement in Phase 3.

Continuous BDTS. Similar to its snapshot counterpart, a continuous BDTS is evaluated by building on top of a continuous DTS algorithm; in practice, the key for this purpose is to additionally consider $span(\cdot,Q)$ and its lower bound $\underline{span}(\cdot,Q)$ in each of the three phases. In particular, both inserting/deleting a trajectory and increasing/decreasing number of results k updates are handled almost identical to continuous DTS as $span(\cdot,Q)$ and $\underline{span}(\cdot,Q)$ for each candidate trajectory in set C are unaffected. A small special case arises when inserting a new trajectory; during Phase 1, the continuous algorithm still treats t_{new} as a full match computing its $dist(t_{new},Q)$ but adds t_{new} to candidate set C only if $span(t_{new},Q)$ does not exceed threshold τ.

On the other hand, updating the set of query points Q may affect the span of the candidate trajectories. When a new query point is added to Q, $\underline{span}(t,Q)$ for each candidate $t \in C$ (which equals $span(t,Q)$ for full matches based on Equation (9)) may increase due to considering an additional term. Under this, during Phase 1, the continuous algorithm needs to examine the contents of set C and potentially eliminate trajectories with $\underline{span}(\cdot,Q) > \tau$. Such an update may force the algorithm to enter Phase 2 as the termination condition of candidate generation may no longer hold. Finally, removing a query point from

Table 1 Experimental parameters

description	parameter	value range	default value
Number of results	k	1, 5, 10, 50, 100	10
Number of query points	$\lvert Q \rvert$	2, 4, 6, 8, 10	6
Span threshold ratio	τ/τ_{min}	1, 1.5, 2, 2.5, 3	3
Linear combination factor	α	0, 0.25,0.5, 0.75, 1	0.5

Q could result in decreasing $\underline{span}(\cdot, Q)$ which means that previously pruned trajectories would now qualify as valid candidates. To guarantee correctness, the continuous algorithms maintain and employ during Phase 1, an additional set C_d with these pruned trajectories, similar to structures N_d and S_d for `SRA`/`SRA+`.

Continuous SDTS. Similar to its snapshot counterpart, a continuous SDTS query intuitively comes as a continuous DTS query where $f(\cdot, Q)$ is considered in place of $dist(\cdot, Q)$. In this spirit, first, the ranking criteria and the lower/upper bounds of $f(\cdot, Q)$ are computed according to Equation (10), and second, similar to BDTS, the $\underline{span}(\cdot, Q)$ lower bound for the candidate trajectories in set C is updated in case of insert/delete query point updates.

Continuous order-aware points-based trajectory search. The continuous DTS, BDTS and SDTS queries can be extended to the case when a visiting order is imposed on the query points, similar to their snapshot counterparts. The ideas discussed in Section 4.3 are directly applicable to address these order-aware variants. For instance, in case of ordered-aware continuous DTS, distance bounds of the candidate trajectories are re-computed according to Equation (11).

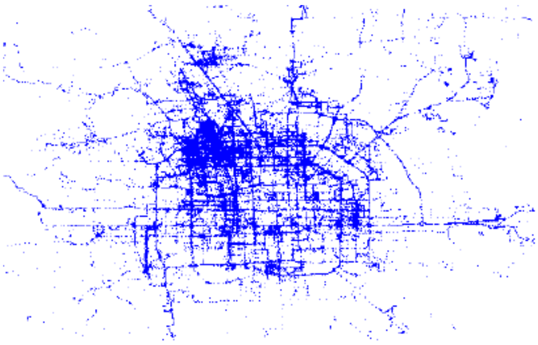
5 Experimental Analysis

We evaluate the efficiency of our methods for both snapshot and continuous points-based trajectory search. Section 5.1 details the setup of our analysis. Sections 5.2 and 5.3 demonstrate the efficiency of our methods for snapshot and continuous points-based trajectory search, respectively. All algorithms were implemented in C++ and the tests run on a machine with Intel Core i7-3770 3.40GHz and 16GB main memory running Ubuntu Linux.

5.1 Setup

We conducted our analysis using real-world trajectories from the GeoLife Project [24–26].[3] The collection contains 17,166 trajectories with approximately 19m points in the city of Beijing, recording a broad range of outdoor movement, from everyday routine to entertainment and sport activities, such

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



(a) GeoLife trajectories

category	cardinality
Restaurants	51,971
Hotels	10,620
Pharmacies	6,963
Schools	6,618
Banks	6,057
Police stations	2,509
Supermarkets	2,356
Gas stations	1,916
Post offices	1,125
Museums	905

(b) Types of POIs

Fig. 4 Experimental dataset in the city of Beijing

as shopping, sightseeing, dining, and cycling; Figure 4(a) illustrates the distribution of the GeoLife trajectories. To generate our query sets, we collected around 90k points of interest (POIs) of various types (see Figure 4(b)), located inside the same area covered by the trajectories. A query set Q is formed by randomly selecting a combination of $|Q|$ types and a particular POI from each type.

We assess the performance of all involved methods measuring their CPU and I/O cost, and the number of candidates they generate over 1,000 distinct query sets Q, while varying (i) the number of returned trajectories k and (ii) the number of query points $|Q|$. In case of BDTS queries, we additionally vary the span threshold via the τ/τ_{min} ratio, where τ_{min} is the minimum possible time required to travel among the query points in Q at a constant velocity of 50km/h. Finally, for SDTS queries, we also vary the weight factor α of Equation (10). Table 1 summarizes all parameters involved in our study

5.2 Snapshot Queries

We report the results of our tests on distance-to-points, span-bounded distance-to-points and span & distance-to-points trajectory snapshot search.

5.2.1 Distance-to-Points Trajectory Search

Figure 5 reports the CPU cost, the I/O cost and the number of generated candidates for the DTS methods. As expected the processing cost of all methods goes up as the values of k and $|Q|$ increase. The tests clearly show that `SRA+` is overall the most efficient evaluation method. We also make the following observations.

First, we observe that `IKNN` always outperforms `GH/QE`; note that this is the first time the methods from [2,18] are compared. Naturally, `GH` comes as the least efficient method; due to the examination order imposed by global heap

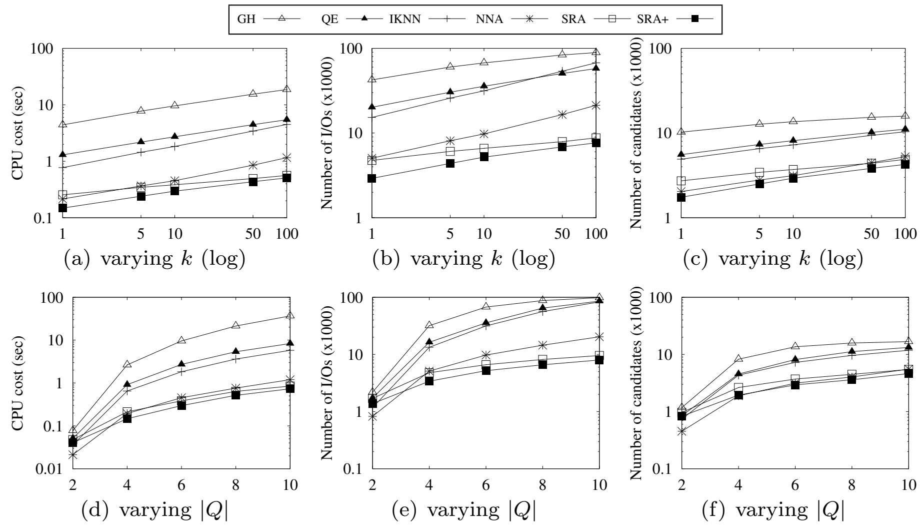


Fig. 5 Performance comparison for Distance-to-Points Trajectory Search

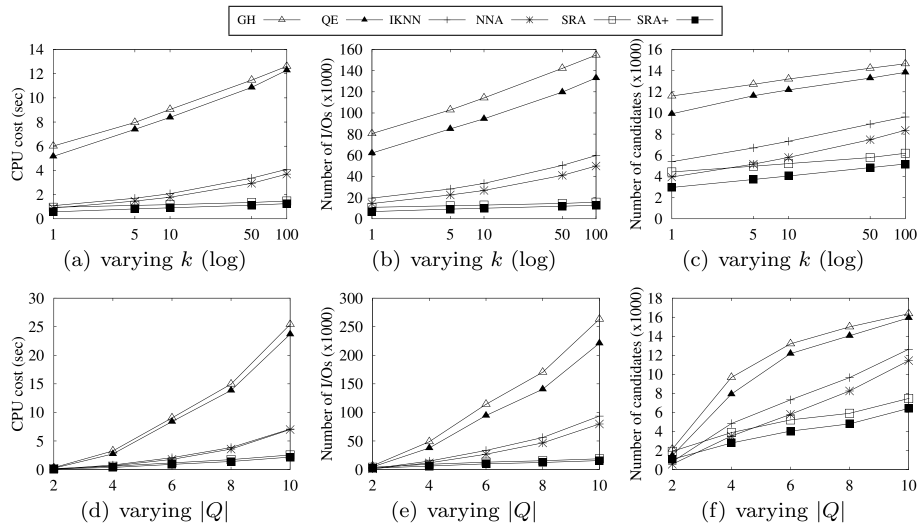


Fig. 6 Performance comparison for Distance-to-Points Trajectory Search (order-aware)

H, the algorithm is unable to cope with the skewed distribution of the real-world data. `QE` manages to overcome the shortcomings of `GH` by completing the empty slots of the most promising candidates. Yet, compared to `IKNN`, `QE` is less efficient due to its weak termination condition for the generation phase; recall that at least k full matches are needed for this purpose which also results in generating a larger number of candidates, as shown in Figures 5(c) and (f). The advantage of `IKNN` over `GH`/`QE` justifies our decision to build the hybrid

NNA method upon the round robin-based candidate generation of IKNN which retrieves nearest neighbor points in batches, and its powerful threshold-based termination condition. NNA is indeed the most efficient NN-based method, in fact with an order of magnitude improvement over IKNN and GH/QE on both CPU and I/O cost. Finally, Figure 5 clearly shows the advantage of the spatial range-based evaluation approach over the NN-based one. SRA is always faster while incurring fewer disk page accesses than IKNN, and in a similar manner, SRA+ outperforms NNA.

We also experimented with the order-aware variant of DTS. Figure 6 depicts similar results to Figure 5; the spatial range-based evaluation approach is again superior to the NN-based and overall, SRA+ is the most efficient method. Nevertheless, it is important to notice that the advantage of completing the most proposing candidates is smaller compared to Figure 5, in terms of the CPU cost. Specifically, observe how close is the running time of GH to QE, of IKNN to NNA and of SRA to SRA+, in Figures 6(a) and (d). This is expected as completing partial matches employs dynamic programming to compute $dist_o(\cdot, Q)$.

5.2.2 Span-Bounded Distance-to-Points Trajectory Search

Next, we investigate the evaluation of BDTS queries while varying k, $|Q|$ and τ/τ_{min}. Based on the findings of the previous section, we use SRA+ as the underlying DTS method. Figure 7 and 8 for the order-aware variant of BDTS, clearly show that ONE–PASS outperforms INCREMENTAL in all cases. As expected, the conservative approach of INCREMENTAL generates a larger number of candidates by performing multiple rounds of generation and refinement which results in both higher running time and more disk page accesses. Last, notice that the evaluation of BDTS becomes less expensive for both methods while increasing τ/τ_{min}, as the number of invalid candidates progressively drops.

5.2.3 Span & Distance-to-Points Trajectory Search

Last, we study the evaluation of SDTS queries. For this experiment, we extended the most dominant method from [2,18], i.e., IKNN, and our methods NNA, SRA and SRA+ following the discussion in Section 4.2. The results in Figure 9 demonstrate, similar to the DTS case in Section 5.2.1, the advantage of both the spatial range-based approach and the SRA+ algorithm which is overall the most efficient evaluation method. Figure 10 show the results for the order-aware variant of SDTS, where the relative performance is identical to Figure 9.

5.3 Continuous Queries

We next switch our focus to continuous points-based trajectory search; in particular, we study the case of continuous DTS queries as detailed in Section 3.

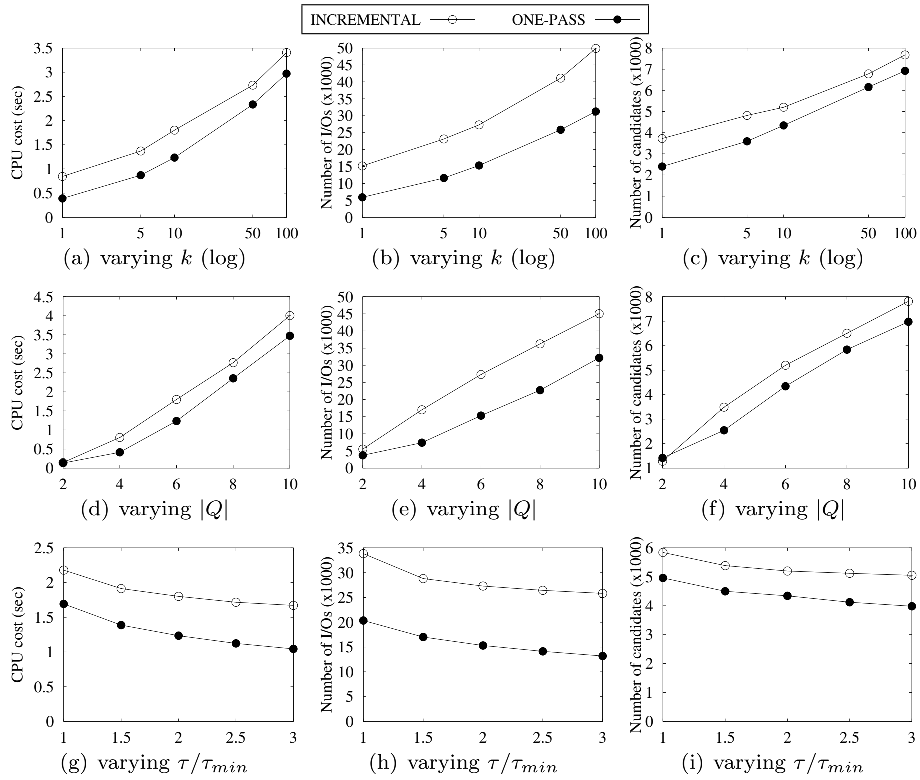


Fig. 7 Performance comparison for Span-Bounded Distance-to-Points Trajectory Search

To this end, the following experimental scenario is considered for each type of updates. An *initial* DTS query is first issued; a DTS algorithm computes and stores the result and the other entries of its state. Then, four updates occur one after the other; the algorithm handles each update event, i.e., it outputs the new result sets. Our previous tests on snapshot DTS queries demonstrated the advantage of our spatial range-based evaluation approach. However, to deliver a complete analysis, we experiment with the continuous counterpart of all GH, QE,IKNN, NNA, SRA and SRA+ methods. For every algorithm, we measure and plot its CPU and I/O cost, and the number of candidate trajectories, regarding both the initial DTS query and each of the updates. Note that we initially set parameters k and $|Q|$ to their default values according to Table 1. Figures 11–13 report the results of our tests using stacked histograms. We make the following observations.

First as expected, the tests back up our original argument about the efficient evaluation of continuous points-based trajectory search. Employing the continuous counterpart of a DTS algorithm which resumes from its previous state is in all cases (algorithms and update types) faster than running the snapshot DTS algorithm from scratch. In the latter case, the cost of handling

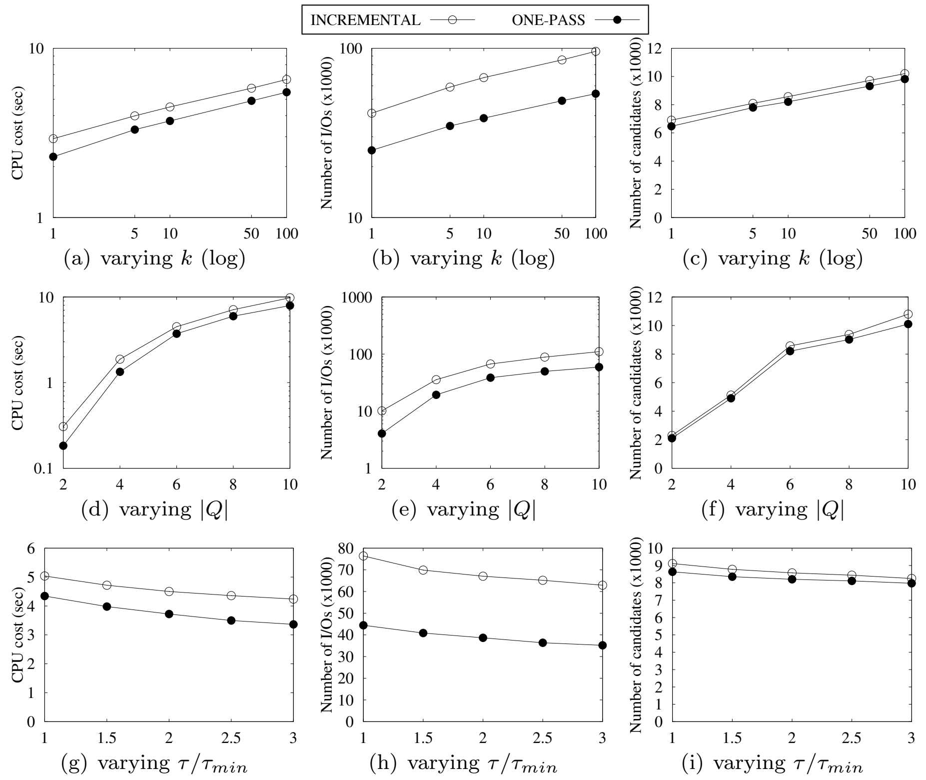


Fig. 8 Performance comparison for Span-Bounded Distance-to-Points Trajectory Search (order-aware)

each update event practically equals the cost of the initial query. We notice that this benefit is larger in case of the `GH`, `IKNN` and `SRA` algorithms compared to `QE`, `NNA` and `SRA+`, which is expected. Recall that all `QE`, `NNA` and `SRA+` periodically fill the empty slots of the most promising partial matches which allows the methods to terminate earlier and drastically reduce the number of candidates. This aggressive approach pays off in case of one-time DTS queries but not under a continuous querying scenario. In fact, the method that benefits the most is `GH`, which in practice, computes such a large number of candidates during the evaluation of the kick-off DTS query that very few additional candidate trajectories need to be identified later while handling the updates.

Second, we observe that the spatial range-based approach outperforms all NN-based approaches, similar to the case of snapshot DTS queries. Notice also that the impact of filling the empty slots of partial matches is less significant for spatial range-based evaluation. In practice, this observation is of great value as it allows us to use a spatial range-based method for both snapshot and continuous DTS queries.

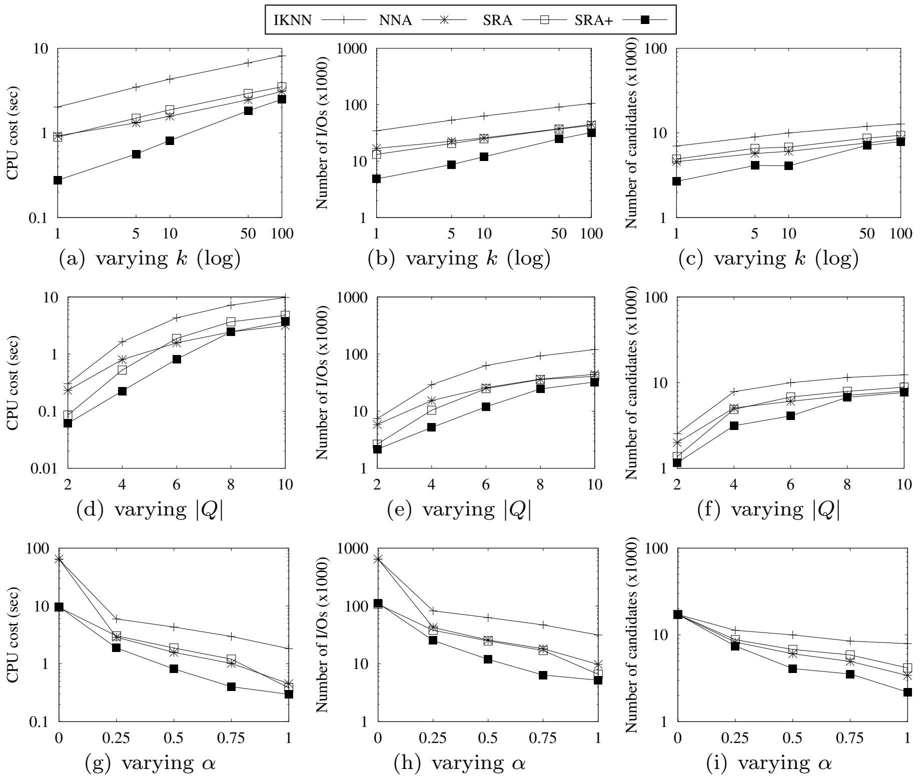


Fig. 9 Performance comparison for Span & Distance-to-Points Trajectory Search

Finally, we also experiment with the order-aware variant of continuous DTS; Figures 14–16 report the results of our tests. As expected, the cost of the order-aware continuous search is (for all algorithms and query types) higher compared to the case when no order is imposed among the query points in set Q. This is because of employing dynamic programming to compute $dist_o(\cdot, Q)$. Nevertheless, our tests still demonstrate the advantage of the continuous counterparts of the DTS algorithms, and most importantly the superiority of the spatial range-based evaluation.

6 Related Work

Apart from the studies [2,18] for distance-to-points search on trajectories detailed in Section 2.2, our work is also related to *top-k* and *nearest neighbor* queries.

Trajectory Search. Previous work in querying trajectories can be classified in three query types [27]. The *P-Query* asks for trajectories satisfying some spatio-temporal relationship to an input point/points. The basic scenario [4]

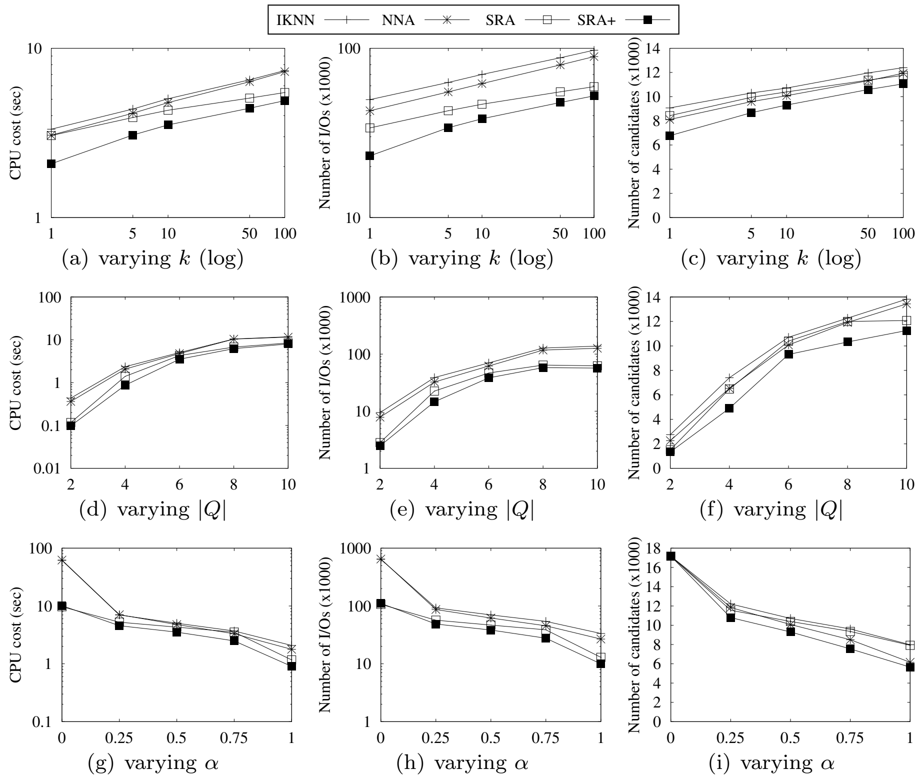
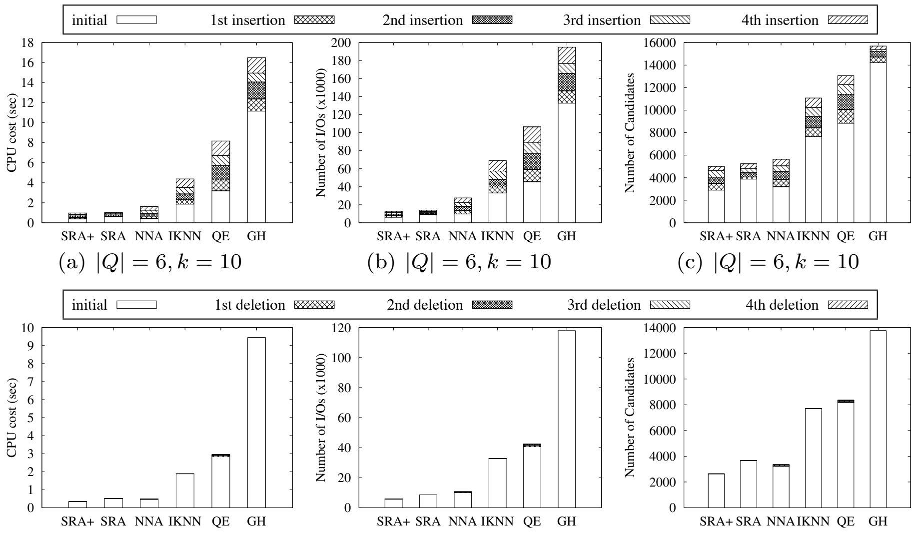


Fig. 10 Performance comparison for Span & Distance-to-Points Trajectory Search (order-aware)

is to find trajectories based on one point ranked by the distance or filtered by a spatio-temporal threshold. The multiple-points trajectory query [2,18] studies a more generic case where the trajectories are retrieved based on their distances to multiple input locations. The DTS problem we study is this paper is an multiple-points P-Query. The *R-Query* searches for trajectory segments belonging to a specified spatio-temporal region. Pfoser et al. [14] study several variants under this category and propose the TB-tree index, which is a hybrid R-tree structure preserving trajectories as well as typical R-tree range searches. Lastly, the *T-Query* asks for trajectories that are similar or close to given trajectories, where a typical application is trajectory clustering [10,11].

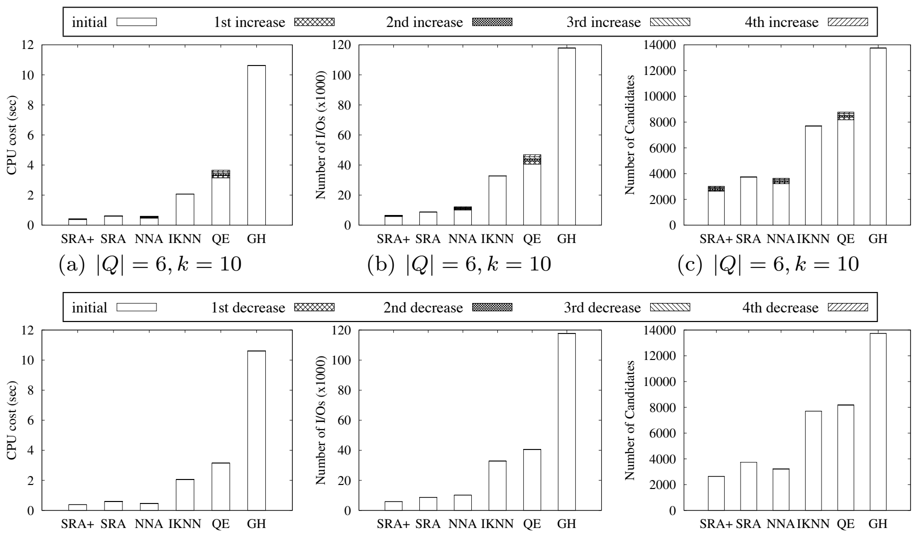
Top-k Queries. Consider a collection of objects, each having a number of scoring attributes, e.g., rankings. Given an aggregate function γ (e.g., SUM) on these scoring attributes, a top-k query returns the k objects with the highest aggregated score. To evaluate such a query, a *sorted* list for each attribute a_i organizes the objects in decreasing order of their value to a_i; requests for *random accesses* of an attribute value based on object identifiers may be also possible. Ilyas et al. overviews top-k queries in [8] providing a categorization of



(a) $|Q| = 6, k = 10$ (b) $|Q| = 6, k = 10$ (c) $|Q| = 6, k = 10$

(d) $|Q| = 6, k = 10$ (e) $|Q| = 6, k = 10$ (f) $|Q| = 6, k = 10$

Fig. 11 Continuous DTS: inserting/deleting four query points.



(a) $|Q| = 6, k = 10$ (b) $|Q| = 6, k = 10$ (c) $|Q| = 6, k = 10$

(d) $|Q| = 6, k = 10$ (e) $|Q| = 6, k = 10$ (f) $|Q| = 6, k = 10$

Fig. 12 Continuous DTS: increasing/decreasing k by 2.

the proposed methods. Specifically, when both sorted and random accesses are possible, the `TA/CA` [3] and `Quick–Combine` [6] algorithms can be applied. `TA` retrieves objects from the sorted lists in a round-robin fashion while a priority queue to organizes the best k objects so far. Based on the last seen attribute values, the algorithm defines an upper score bound for the unseen objects, and

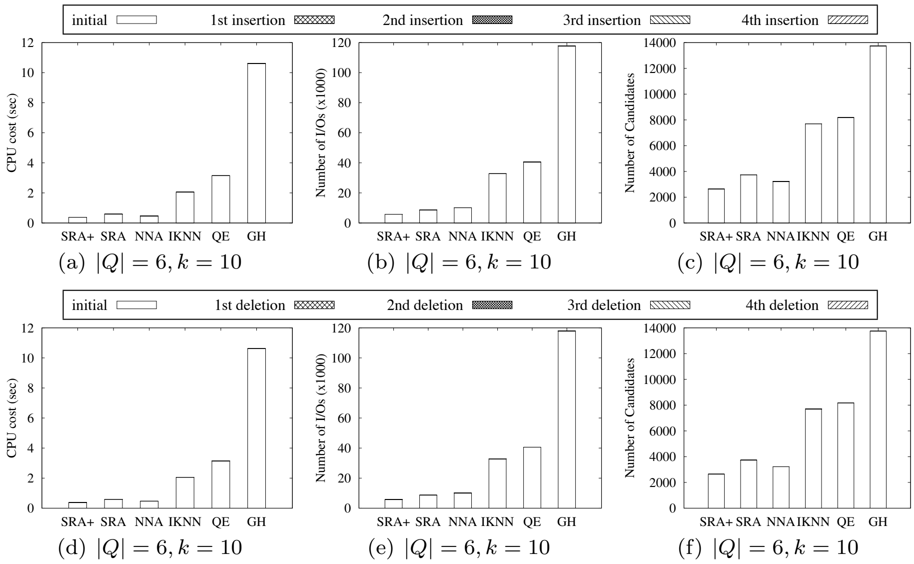


Fig. 13 Continuous DTS: inserting/deleting four trajectories.

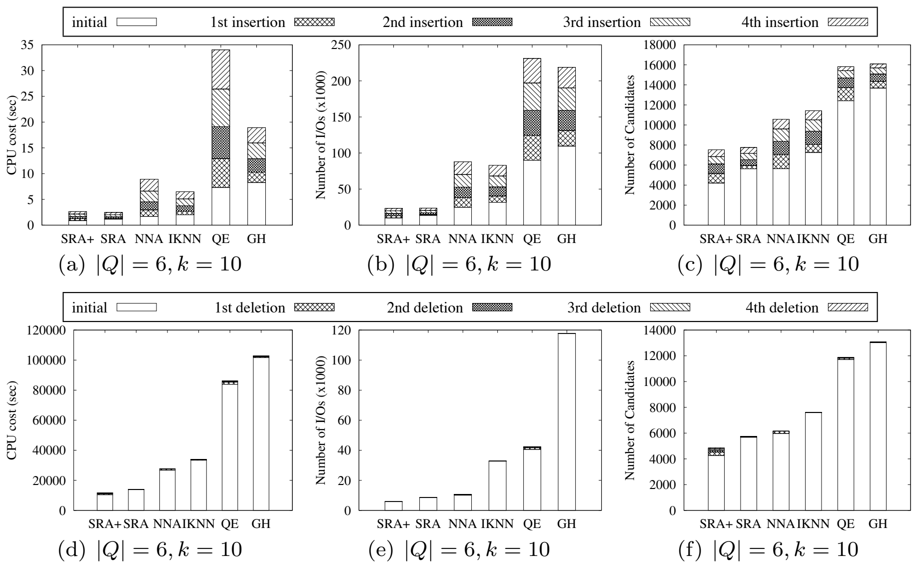
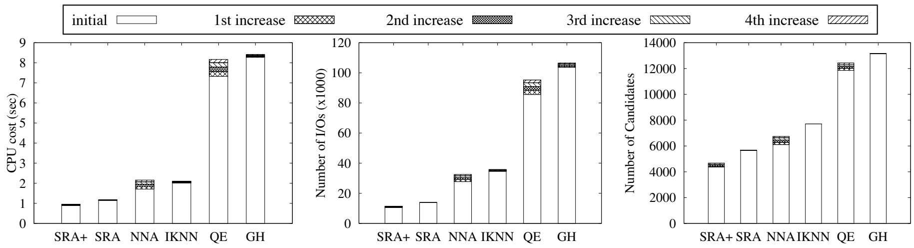
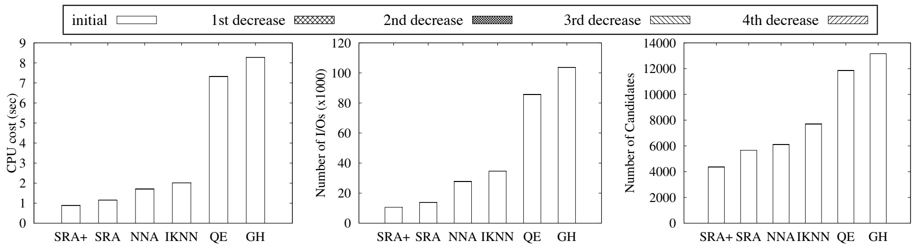


Fig. 14 Continuous DTS: inserting/deleting four query points (order-aware).

terminates if current k-th highest aggregate score is higher than this threshold. `TA` assumes that the costs of the two different access methods are the same. As an alternative, `CA` defines a ratio between these costs to control the number of random accesses, which in practice are usually more expensive than sorted accesses. Hence, the algorithm periodically performs random accesses to col-

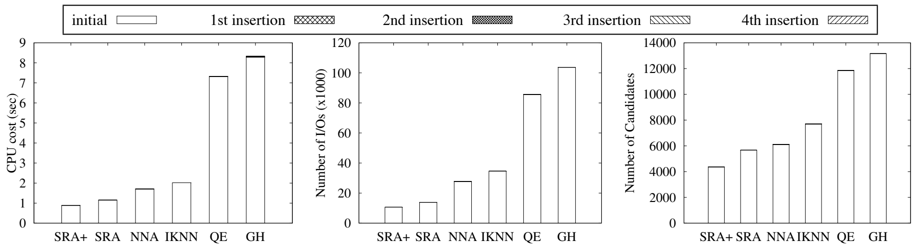


(a) $|Q| = 6, k = 10$ (b) $|Q| = 6, k = 10$ (c) $|Q| = 6, k = 10$

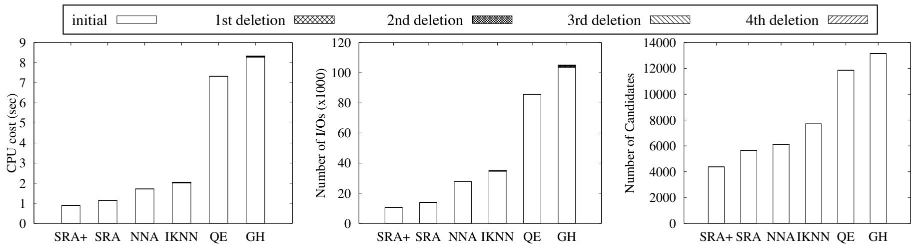


(d) $|Q| = 6, k = 10$ (e) $|Q| = 6, k = 10$ (f) $|Q| = 6, k = 10$

Fig. 15 Continuous DTS: increasing/decreasing k by 2 (order-aware).



(a) $|Q| = 6, k = 10$ (b) $|Q| = 6, k = 10$ (c) $|Q| = 6, k = 10$



(d) $|Q| = 6, k = 10$ (e) $|Q| = 6, k = 10$ (f) $|Q| = 6, k = 10$

Fig. 16 Continuous DTS: inserting/deleting four trajectories (order-aware).

lect unknown values for the most "promising" objects. Last, the idea behind `Quick–Combine` is to favor accesses from the sorted lists of attributes which significantly influence the overall scores and the termination threshold. In contrast, when only sorted accesses are possible, the `NRA` [3] and `Stream–Combine` [5] algorithms can be applied. Intuitively, `Stream–Combine` operates similar to

`Quick–Combine` without performing any random accesses. In Section 2.2, we discuss how the methods in [2,18] build upon previous work on top-k queries to address distance-to-points search on trajectories.

Nearest Neighbor Queries. There is an enormous amount of work on the *nearest neighbor* (NN) query (also known as similarity search), which returns the object that has the smallest distance to a given query point; k-NN queries output the k nearest objects in ascending distance. Roussopoulos et al. proposed a depth-first approach to k-NN query in [16] while Hjaltason et al. enhanced the evaluation with a best-first search strategy in [7]. An overview of index-based approaches can be found in [1]; efficient methods for metric spaces, e.g., [9], and high-dimensional data, e.g., [19], have also been proposed.

For a set of query points, the *aggregate nearest neighbor* (ANN) query [13] retrieves the object that minimizes an aggregate distance to the query points. As an example, for the MAX aggregate function and assuming that the set of query points are users, and distances represent travel times, ANN outputs the location that minimizes the time necessary for all users to meet. In case of the SUM function and Euclidean distances, the optimal location is also known as the Fermat-Weber point, for which no closed-form formula exists.

In continuous, or long-standing, NN queries, the result of a standard NN query must be continuously maintained as updates to the query location and/or the data objects appear. The methods in [17] and [22] handle the case when only the query location is moving. The former retrieves $m > k$ nearest neighbors hoping that the result at a future time is among these m objects, provided that the query does not move much. The latter returns a Voronoi-based validity region such that the result does not change as long as the query remains within the region. [21], [20] and [12] present incremental grid-based methods for general continuous monitoring NN queries, i.e., when all objects move in a non-predictive manner; the last two works feature shared execution techniques to handle multiple NN queries.

7 Conclusions

In this paper, we studied the efficient evaluation of points-based trajectory search. After revisiting the existing methods (`IKNN` and `GH/QE`) for distance-to-points search, which examine the trajectories in ascending order of their distance to the queries points, we devised a hybrid algorithm which outperforms them by a wide margin. Then, we proposed a spatial range-based approach; our experiments on real-world trajectories showed that this approach outperforms any NN-based method. Besides improving the performance of distance-to-points search, we also introduced and investigated the evaluation of a practical variant for points-based trajectory search, which also takes into account the temporal aspect of the trajectories, as well as the order-aware case of a query point sequence. Moreover, we introduced and studied the continuous case of all points-based trajectory search variants, where the goal is to maintain the result set as updates to the query parameters and/or the trajectories appear. As a direction for future work, we plan to consider additional types of annotated data on the trajectories in points-based search, such as textual and social information.

References

1. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3), 322–373 (2001)
2. Chen, Z., Shen, H.T., Zhou, X., Zheng, Y., Xie, X.: Searching trajectories by locations: an efficiency study. In: SIGMOD, pp. 255–266 (2010)
3. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS, pp. 102–113 (2001)
4. Frentzos, E., Gratsias, K., Pelekis, N., Theodoridis, Y.: Algorithms for nearest neighbor search on moving object trajectories. GeoInformatica **11**(2), 159–193 (2007)
5. Güntzer, U., Balke, W., Kießling, W.: Towards efficient multi-feature queries in heterogeneous environments. In: ITCC, pp. 622–628 (2001)
6. Güntzer, U., Balke, W.T., Kießling, W.: Optimizing multi-feature queries for image databases. In: VLDB, pp. 419–428 (2000)
7. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999)
8. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4) (2008)
9. Jagadish, H.V., Ooi, B.C., Tan, K., Yu, C., Zhang, R.: iDistance: An adaptive b^{+}-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. **30**(2), 364–397 (2005)
10. Lee, J.G., Han, J., Whang, K.Y.: Trajectory clustering: A partition-and-group framework. In: SIGMOD, pp. 593–604 (2007)
11. Li, X., Han, J., Lee, J.G., Gonzalez, H.: Traffic density-based discovery of hot routes in road networks. In: SSTD, pp. 441–459 (2007)
12. Mouratidis, K., Hadjieleftheriou, M., Papadias, D.: Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In: SIGMOD, pp. 634–645 (2005)
13. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Syst. **30**(2), 529–576 (2005)
14. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches to the indexing of moving object trajectories. In: VLDB, pp. 395–406 (2000)
15. Qi, S., Bouros, P., Sacharidis, D., Mamoulis, N.: Efficient point-based trajectory search. In: SSTD, pp. 179–196 (2015)
16. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD, pp. 71–79 (1995)
17. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. In: SSTD, pp. 79–96 (2001)
18. Tang, L.A., Zheng, Y., Xie, X., Yuan, J., Yu, X., Han, J.: Retrieving k-nearest neighboring trajectories by a set of point locations. In: SSTD, pp. 223–241 (2011)
19. Tao, Y., Yi, K., Sheng, C., Kalnis, P.: Quality and efficiency in high dimensional nearest neighbor search. In: SIGMOD, pp. 563–576 (2009)
20. Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE, pp. 643–654 (2005)
21. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE, pp. 631–642 (2005)
22. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD, pp. 443–454 (2003)
23. Zheng, Y.: Trajectory data mining: An overview. ACM Trans. Intell. Syst. and Tech. (2015)
24. Zheng, Y., Li, Q., Chen, Y., Xie, X., Ma, W.: Understanding mobility based on GPS data. In: UbiComp, pp. 312–321 (2008)

25. Zheng, Y., Xie, X., Ma, W.: Geolife: A collaborative social networking service among user, location and trajectory. IEEE Data Eng. Bull. **33**(2), 32–39 (2010)
26. Zheng, Y., Zhang, L., Xie, X., Ma, W.Y.: Mining interesting locations and travel sequences from gps trajectories. In: WWW, pp. 791–800 (2009)
27. Zheng, Y., Zhou, X. (eds.): Computing with Spatial Trajectories. Springer (2011)