4-2010

Algorithms for Constrained k-Nearest Neighbor Queries over Moving Object Trajectories

Yunjun GAO
Singapore Management University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Gencai CHEN

Qing LI

Chun CHEN

DOI: https://doi.org/10.1007/s10707-009-0084-5

Citation

GAO, Yunjun; ZHENG, Baihua; CHEN, Gencai; LI, Qing; and CHEN, Chun. Algorithms for Constrained k-Nearest Neighbor Queries over Moving Object Trajectories. (2010). *Geoinformatica*. 14, (2), 241-276. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/1984

Published in GeoInformatica, April 2010, Volume 14, Issue 2, pp 241-276.
http://dx.doi.org/10.1007/s10707-009-0084-5

Algorithms for constrained *k*-nearest neighbor queries over moving object trajectories

Yunjun Gao · Baihua Zheng · Gencai Chen · Qing Li

Abstract An important query for spatio-temporal databases is to find nearest trajectories of moving objects. Existing work on this topic focuses on the closest trajectories in the whole data space. In this paper, we introduce and solve *constrained k-nearest neighbor* (C*k*NN) queries and *historical continuous* C*k*NN (HCC*k*NN) queries on R-tree-like structures storing historical information about moving object trajectories. Given a trajectory set *D*, a query object (point or trajectory) *q*, a temporal extent *T*, and a constrained region *CR*, (i) a C*k*NN query over trajectories retrieves from *D* within *T*, the *k* ($\geq$ 1) trajectories that lie closest to *q* and intersect (or are enclosed by) *CR*; and (ii) an HCC*k*NN query on trajectories retrieves the constrained *k* nearest neighbors (C*k*NNs) of *q at any time instance* of *T*. We propose a suite of algorithms for processing C*k*NN queries and HCC*k*NN queries respectively, with different properties and advantages. In particular, we thoroughly investigate two types of C*k*NN queries, i.e., C*k*NN$_P$ and C*k*NN$_T$, which are defined with respect to stationary query points and moving query trajectories, respectively; and two types of HCC*k*NN queries, namely, HCC*k*NN$_P$ and HCC*k*NN$_T$, which are continuous

This work is an extended version of [14].

Y. Gao (✉) · B. Zheng
School of Information Systems, Singapore Management University, 80 Stamford Road,
Singapore 178902, Singapore
e-mail: yjgao@smu.edu.sg
e-mail: gaoyj@zju.edu.cn

B. Zheng
e-mail: bhzheng@smu.edu.sg

Y. Gao · G. Chen
College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027,
People's Republic of China

G. Chen
e-mail: chengc@zju.edu.cn

Q. Li
Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong
e-mail: itqli@cityu.edu.hk

counterparts of C*k*NN$_P$ and C*k*NN$_T$, respectively. Our methods utilize an existing data-partitioning index for trajectory data (i.e., TB-tree) to achieve low I/O and CPU cost. Extensive experiments with both real and synthetic datasets demonstrate the performance of the proposed algorithms in terms of efficiency and scalability.

Keywords Query processing · Nearest neighbor · Moving object trajectory · Algorithm

1 Introduction

With the advances of wireless communication, mobile computing, and positioning technologies, it has become possible to obtain and manage (e.g., model, index, query, etc.) the trajectories of moving objects in real life. Trajectory analysis is an important building block of many applications. For instance, it is very useful for zoologists to determine the living habits and migration patterns of certain groups of animals by mining the motion trajectories of animals in a large natural protection area. In a city traffic monitoring system, analyzing the trajectories of existing vehicles may help decision-makers locate popular routes. Therefore, the *k*-nearest neighbor (*k*NN) search for moving object trajectories will be very useful for the above applications. It retrieves from a set of trajectories within a predefined temporal extent, the *k* (≥ 1) trajectories that are closest to a given query object. Consider, for example, Fig. 1(a) where the dataset consists of 5 trajectories of animals, labeled as Tr_1, Tr_2, Tr_3, Tr_4, Tr_5, in a 3D space (two dimensions for spatial positions, and one for time). In this diagram, Tr_2 and Tr_3 are the 2 nearest neighbors (NNs) of a specified query object q inside the time interval $[t_i, t_j]$.

In some real examples, however, users may only have interests in those trajectories in a spatially constrained area and enforce constrained regions on *k*NN queries for trajectory data. For example, assuming that the trajectories of animals over a long time period are known in advance, the zoologists may pose the following query: *find the two closest animal's trajectories in a restricted region (e.g., the rectangle area that locates in the east of the lab) to a specified query object (e.g., lab, food source, etc.) within the time period* $[t_i, t_j]$. Figure 1(b) illustrates a case in which the shadowed rectangle denotes the constrained region, and $\{Tr_3, Tr_4\}$ is the query result. Note that the result of the *2*NN search without region restriction would be $\{Tr_2, Tr_3\}$, as shown in Fig. 1(a).

Given a trajectory set D of moving objects, a query object (point or trajectory) q, a time extent T, and a constrained region CR, a C*k*NN query over trajectories retrieves from D

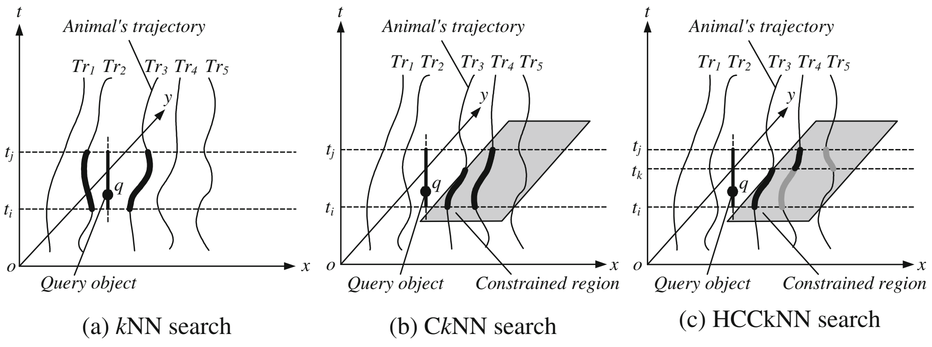


Fig. 1 Example of *k*NN, C*k*NN, and HCC*k*NN queries on moving object trajectories for *k*=2

within T, the k trajectories that lie closest to q and meanwhile cross (or fully fall into) the area bounded by CR. Like constraint NN search over point objects [9], CkNN is to apply conventional (i.e., unconstrained) kNN retrieval inside a specified region. In addition, it might be used as an off-the-shelf component for those divide-and-conquer algorithms that, in order to improve the search performance, partition the entire search space into disjoint cells and perform kNN search in each cell independently.

Conventional kNN queries for spatial and spatiotemporal objects that do not consider constraints have been studied extensively (as to be surveyed in Section 2.2). The existing algorithms can be divided into three categories, based on the fact that whether the query points and/or data objects are moving. They are (i) static kNN query for static objects [5, 6, 16, 19, 29, 31], (ii) moving kNN query for static objects [33, 35, 36, 41], and (iii) moving kNN query for moving objects [3, 4, 17, 18, 22–24, 28, 39, 40]. Recently, kNN search on moving object trajectories has also been addressed [10–13]. However, to the best of our knowledge, none of the existing work has examined the CkNN query over moving object trajectories, which is an interesting problem from the research point of view. First, compared with conventional kNN search on trajectory data, CkNN retrieval over trajectory data is more challenging as it needs consider the proximity between the trajectory and the query object and the constrained region whose size and shape might be arbitrary. Second, it is different from the constrained NN search on spatial objects [9]. For a given query point and a spatial region, constrained NN search returns the closest data points, among those are inside the specified region, to the query point. However, CkNN retrieval over moving object trajectories considers both the spatial and temporal components of trajectories and meanwhile support those queries issued at a query trajectory. Therefore, the algorithms presented in [9] cannot be directly applied to our problem.

In this paper, we study CkNN queries over moving object trajectories and develop several algorithms[1]. In particular, we thoroughly investigate two types of CkNN queries, termed as CkNN$_\mathrm{P}$ and CkNN$_\mathrm{T}$ queries, which are defined with respect to stationary query points and moving query trajectories, respectively. Our approaches are based on existing R-tree-like structures storing historical information about moving object trajectories (i.e., TB-tree [27]) in order to achieve low I/O cost and CPU overhead. Our solutions explore both the two-step processing framework and the single-step processing framework. The former employs range queries and kNN queries sequentially which might incur multiple scanning of the underlying index structure, while the latter integrates those two steps into a single traversal of the index.

In addition, we extend our methodology to *historical continuous constrained k-nearest neighbor* (HCCkNN) search over moving object trajectories, which retrieves from D the constrained k nearest neighbors (CkNNs) of a given query object *q at any time instance* of the specified time period T. Specifically, the output of an HCCkNN query comprises k lists, with i-th ($1 \le i \le k$) list containing a set of $\langle Tr, [t_i, t_j)\rangle$ tuples. Here, Tr is a trajectory in D (i.e., $Tr \in D$), and $[t_i, t_j)$ is the time extent (within T) during which Tr is the i-th NN of q. In Fig. 1(c), for instance, the 1st list includes $\{\langle Tr_3, [t_i, t_k)\rangle, \langle Tr_4, [t_k, t_j]\rangle\}$, which means that

[1] A preliminary work has been published in DASFAA'08 as a short paper [14], in which the concept of constrained kNN (CkNN) search over moving objects trajectories has been introduced. However, due to the space limitation, we only managed to present the basic idea, but not the details, of CkNN query processing. In addition, the concept of historical continuous constrained kNN search on moving objects trajectories has not been defined in that work.

trajectory Tr_3 is the 1st NN for the sub-interval $[t_i, t_k)$ and trajectory Tr_4 is the 1st NN within the interval $[t_k, t_j]$; and the 2nd list contains $\{\langle Tr_4, [t_i, t_k)\rangle, \langle Tr_5, [t_k, t_j)\rangle\}$, indicating that trajectory Tr_4 is the 2nd NN within the interval $[t_i, t_k)$ and trajectory Tr_5 is the 2nd NN within the interval $[t_k, t_j]$. The HCC*k*NN retrieval is actually a continuous counterpart of the C*k*NN query. Correspondingly, we also explore two types of HCC*k*NN queries, called HCC*k*NN$_P$ and HCC*k*NN$_T$ queries which are continuous counterparts of C*k*NN$_P$ and C*k*NN$_T$ queries, respectively. To sum up, the key contributions of this paper are as follows:

- We identify and formally define the C*k*NN search and the HCC*k*NN retrieval for moving object trajectories, respectively.
- We propose a suite of algorithms to efficiently tackle the C*k*NN retrieval on moving object trajectories. In particular, we present two-step (including NN search followed by a range query and range query followed by NN search) and one-step (including depth-first and best-first) algorithms for processing such queries, using TB-tree, a variant of R-tree, as the underlying index structure. Moreover, we extend our methods to answer HCC*k*NN search over moving object trajectories as well.
- We conduct extensive experiments with both real and synthetic datasets under various settings to evaluate the performance of our proposed algorithms in terms of efficiency and scalability.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 gives formal definitions for C*k*NN and HC*k*NN queries. In Sections 4 and 5, we discuss the algorithms for C*k*NN$_P$ and C*k*NN$_T$ queries respectively, and describe the algorithms for HCC*k*NN$_P$ and HCC*k*NN$_T$ queries in Section 6 and 7 respectively. Section 8 presents the performance evaluation of the proposed algorithms and reports our findings. Finally, Section 9 concludes the paper with some directions for future work.

2 Related work

In this section, we review existing work related to C*k*NN. We first briefly discuss access methods for historical trajectories of moving objects in Section 2.1, and then survey previous work on *k*NN queries in spatial and spatio-temporal databases in Section 2.2.

2.1 Indexing of moving object trajectories

The trajectory of a moving object is the path it takes along time. Therefore, trajectories can describe the motion of objects in a 2D/3D space and be considered as 2D/3D time series data. Here, we only concentrate on R-tree-like structures [20] that store historical information about moving object trajectories such as 3DR-tree [38], TB-tree [27], STR-tree [27], and MV3R-tree [34]. Specifically, 3DR-tree [38] treats time as an extra dimension in addition to two spatial dimensions and it supports both range and time slice queries. STR-tree [27] is an extension of the R-tree. It takes spatial closeness and partial trajectory preservation into account. TB-tree [27] extends the STR-tree to process trajectories. It can deal with both *trajectory-based queries* and traditional spatial queries efficiently. MV3R-tree [34] maintains two trees, an MVR-tree to process time-slice queries, and a small auxiliary 3DR-tree built on the leaf nodes of the MVR-tree to process long interval queries. It can efficiently handle both time-slice and time-interval queries by taking advantage of both the MVR-tree and the 3DR-tree. A good survey of the access methods for trajectory data can be found in [21]. In our study, we assume that the dataset is indexed by a TB-tree due to its high efficiency in trajectory-based queries. The structure of TB-tree is outlined as follows.

The TB-tree emphasizes *trajectory preservation*. Thus, each leaf node in the tree contains only line segments belonging to the same trajectory, and is of the form (*id*, *MBB*, *Orientation*), where *id* is the identifier of 3D trajectory segment (considering time as one dimension), *MBB* denotes the minimum bounding box of the 3D line segment, and *Orientation* whose value varies between 1 and 4 specifies how the 3D line segment is enclosed within the *MBB*. All the leaf nodes containing the same trajectory segments are connected by a doubly linked list. This structure strictly preserves trajectory evolution and greatly improves the performance of trajectory-based query processing. A partial example TB-tree for a trajectory is depicted in Fig. 2.

2.2 *k*NN queries in spatial and spatio-temporal databases

In the past decade, numerous algorithms for *k*NN (and NN) queries have been proposed in the database literature. According to the assumptions on whether the query points and/or data objects are moving, the existing algorithms can be divided into three categories. The first category assumes both the query point and the data objects are static, and most of the algorithms fallen in this category follow either *depth-first* (DF) [6, 29] or *best-first* (BF) [16] traversal paradigm. DF algorithm [6, 29] traverses the tree in the depth-first fashion according to some distance metrics such as *mindist* and *minmaxdist*, and develops several pruning heuristics to prune the search space. DF algorithm is simple but it is suboptimal in terms of I/O, i.e., it accesses more nodes than necessary, as demonstrated in [26]. Motivated by this, the BF algorithm [16] tries to minimize the access of unnecessary nodes. It employs a priority queue to order the entries visited so far by their minimum distances to a given query point (i.e., *mindist*). Although BF achieves optimal I/O performance, it suffers from buffer thrashing if the heap becomes larger than the available memory. Both BF and DF are based on the R-tree [15] or its variants [2, 32] of the data objects. Alternatively, a solution-based approach is proposed to pre-compute the Voronoi cells for all the data objects and

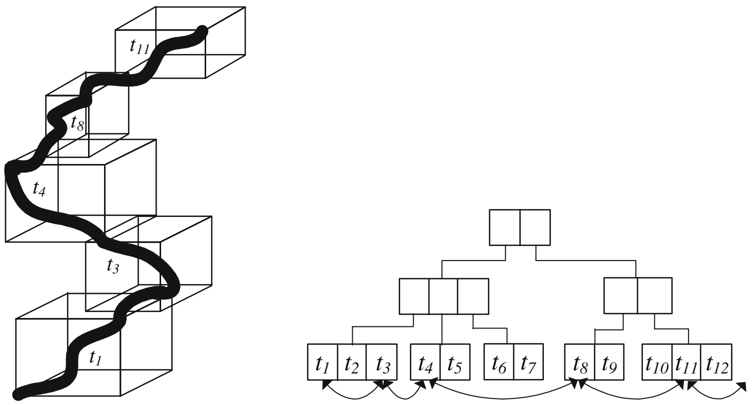

t_{11}
t_8
t_4
t_3
t_1
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
(a) A trajectory
(b) A part of the corresponding TB-tree

Fig. 2 Example of a trajectory and the corresponding TB-tree structure

covert the NN search into a point location problem. In [5], an NN query processing algorithm based on Voronoi cells is proposed, and a multi-step algorithm for *k*NN search is proposed in [19]. However, the *k*NN search generates many intermediate candidates during query processing and hence suffers from a poor performance, as shown in [31]. Based on this observation, Seidl and Kriegel [31] develop an optimal multi-step algorithm for *k*NN retrieval to minimize the number of candidates that are retrieved from the underlying index.

The second category assumes that the query point is moving while the data objects are static, e.g., continuous nearest neighbor (CNN) search. The first attempt to handle CNN is proposed in [33]. It utilizes a periodical sampling technique to repeatedly perform traditional NN queries at some predefined sampling points of a given query line segment. Thereafter, a tight range, based on those NN objects obtained at previous sampling step, is approximated to bound all the possible answers. However, its performance highly depends on the number and positions of those sampling points. Specifically, a small number of sampling points increase the performance but may result in incorrect results, whereas a large number of sampling points create significant computational overhead but decrease the possibility of false misses. In order to conduct exact searches, Tao and Papadias [35, 36] develop two CNN query processing algorithms using R-trees as the underlying data structure. The first algorithm is based on the concept of *Time-Parameterized* (TP) queries, which treats a query line segment as the moving trajectory of a query point [35]. Thus, the nearest object to the moving query point is valid only for a limited duration and a new TP query is issued to retrieve the next nearest object once the valid time of the current query expires, i.e., when a split point is reached. Although the TP approach avoids the drawbacks of sampling, the performance depends on the number of answer objects *m*, as it needs issue *m* TP queries. In order to improve the performance, the second algorithm solves the CNN query problem by applying a single query to retrieve all the answer points for the whole query line segment [36]. Consequently, only a single navigation of R-tree is incurred.

The last category assumes both the query point and the data objects are moving. Specifically, Kollios et al. [18] use dual transformation to tackle NN queries for moving objects in 1D space. The method can determine the one object that comes closer to the query during a predefined time interval $[t_s, t_e]$, but not the *k* NNs for every time instance of $[t_s, t_e]$. Benetis et al. [3, 4] first develop an algorithm to process the NN search, and then extend their approaches to support *k*NN search, for continuously moving points. Tao and Papadias [35] propose a technique, called *time-parameterized queries*, which can be applied with mobile queries, mobile objects or both, given an appropriate indexing structure. Iwerks *et al.* [17] investigate the problem of continuous *k*NN queries for moving points to allow the motion functions of the points to be changed. Raptopoulou et al. [28] present efficient methods for NN query processing on moving-query, moving-object case, using a TPR-tree [30] as an underlying index structure. Recently, the CNN *monitoring* problem has also been studied in the literature. These include (i) CNN *monitoring* in the Euclidean space [22, 39, 40], (ii) CNN *monitoring* in road network [24], and (iii) CNN *monitoring* in a distributed environment [23].

Even though *k*NN queries have been well-studied in the field of spatial databases, the *k*NN query for moving object trajectories is still a relatively new research problem. It is first investigated by Frentzos et al. [10, 11]. Several search algorithms based on R-tree-like structures that store historical information about moving object trajectories are proposed, varying with respect to the type of the query objects (points or trajectories) and the type of the query result (historical continuous or not). In particular, they develop a series of DF (and BF) based algorithms for non-continuous *k*NN (and NN) queries as well as DF based algorithms for their continuous counterparts. In our earlier work [12, 13], we have proposed several efficient BF based algorithms for answering snapshot and continuous *k*NN queries over moving object trajectories. However, all the above work does not consider any spatial constraint.

Furthermore, a large number of variants of *k*NN (and NN) queries (e.g., *constrained* NN search [9], *group* NN search [25], *all* NN search [42], *surface* *k*NN search [8], etc.) have been examined as well. Thereinto, constrained NN search is related to our work. In [9], Ferhatosmanoglu *et al.* introduce and solve the constrained NN retrieval for spatial objects, which discovers the NN(s) in a restricted area of the data space. As mentioned earlier, however, this problem differs from our study in this paper, since it does not take the temporal information of objects and query trajectory input into consideration.

3 Problem statement

The problem that we are going to focus on in this paper is formulized in this section. In order to facilitate the following discussion, Table 1 lists the notations used frequently.

Let $D = \{Tr_1, Tr_2, \ldots, Tr_n\}$ be a set of trajectories corresponding to n moving objects. The trajectory Tr_i ($1 \le i \le n$) of a moving object i is represented as a sequence of trajectory segments in the form of $[((s_{i1\text{-}start}, t_{i1\text{-}start}), (s_{i1\text{-}end}, t_{i1\text{-}end})), ((s_{i2\text{-}start}, t_{i2\text{-}start}), (s_{i2\text{-}end}, t_{i2\text{-}end})) \ldots, ((s_{im\text{-}start}, t_{im\text{-}start}), (s_{im\text{-}end}, t_{im\text{-}end}))]$. Here, m is the number of trajectory segments contained in Tr_i, and $s_{ij\text{-}start}$ and $s_{ij\text{-}end}$ specify 2D position vectors that are sampled at timestamp $t_{ij\text{-}start}$ and $t_{ij\text{-}end}$ respectively. We refer to the rectangle bounded by both $s_{ij\text{-}start}$ and $s_{ij\text{-}end}$ as a *spatial bound* of trajectory Tr_i, denoted by $Tr_i.spatialbound$. Since we focus on the queries on historical trajectories of moving objects, $t_0 \le t_{ij\text{-}start} < t_{ij\text{-}end} \le t_{now}$ with t_0 representing the beginning of the calendar and t_{now} denoting the current time point.

Table 1 Symbols used in this paper

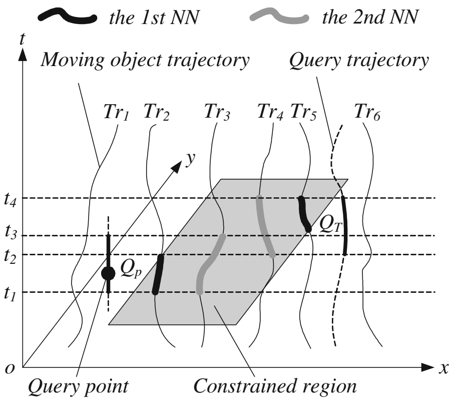
Symbol	Description
Tr	a moving object trajectory
$Tr(t)$	the point location along the trajectory at time point t
D	a set of moving object trajectories Tr_i
k	the number of requested nearest neighbors
q	a query object (either Q_P or Q_T)
Q_P	a query point
Q_T	a query trajectory
$Q_T(t)$	the point location along the query trajectory Q_T at time point t
T	a query time interval in the form of $(T.t_s, T.t_e)$
CR	a constrained region
S_{rslt}	the set of query result for C*k*NN retrieval

Recall that in this paper, our ultimate goal is to provide efficient support for C*k*NN search and the HCC*k*NN search over moving object trajectories. Without loss of generality, we assume that *CR* is in a rectangular shape and those *CR* in arbitrary shapes can be bounded by minimum bounding boxes. Moreover, we distinguish two kinds of query objects, i.e., Q_P and Q_T, in our study. As mentioned in the previous discussion, we investigate two types of C*k*NN queries, i.e., C*k*NN$_\mathrm{P}$ and C*k*NN$_\mathrm{T}$ queries, and two types of HCC*k*NN queries, i.e., HCC*k*NN$_\mathrm{P}$ and HCC*k*NN$_\mathrm{T}$ queries, with respect to Q_P and Q_T respectively. The detailed definitions of these queries are defined as follows.

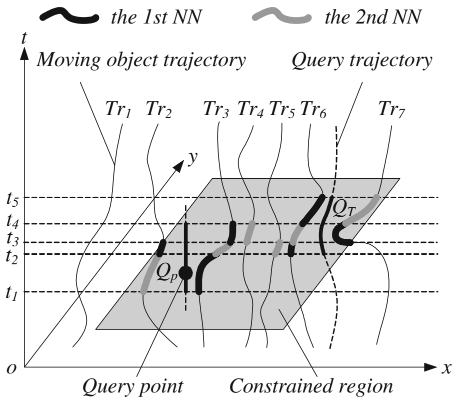
Definition 1 (Distance metric *MinDist* (*Tr, q, T*)) *Given Tr, q (either Q_P or Q_T), and T, the Euclidean distance between Tr and q within T, denoted by MinDist (Tr, q, T), is defined as the minimal Euclidean distance from a point along Tr during T to a point from q inside T, i.e., MinDist (Tr, q, T) = Min{dist (p_t, q_t) | ∀ t ∈ T, p_t = Tr(t), q_t = Q_P if q is a point Q_P* or *q_t = $Q_T(t)$* if *q is a trajectory}, where dist (p, q) represents the Euclidean distance between two objects p and q.*

Definition 2 (C*k*NN query over moving object trajectories) *Given D, q (either Q_P or Q_T), T, CR, and k, a constrained k-nearest neighbor (CkNN) query with respect to q, denoted as CkNN$_P$ and CkNN$_T$ respectively, retrieves from D during T, a set S_{rslt} of k moving object trajectories such that all the trajectories in S_{rslt} are closest to q and intersect (or are enclosed by) CR, i.e., ∀ Tr ∈ S_{rslt}, ∀ Tr′ ∈ {Tr′ ∈ D ∧ Tr′ ∩ CR≠∅} − S_{rslt}, MinDist (Tr, q, T) ≤ MinDist (Tr′, q, T).*

Consider, for example, Fig. 3(a) illustrates a C*2*NN$_\mathrm{P}$ retrieval on $D = \{Tr_1, Tr_2, Tr_3, Tr_4, Tr_5, Tr_6\}$ within $T = [t_1, t_3]$. Trajectories Tr_2 and Tr_3 form $S_{rslt} = \{Tr_2, Tr_3\}$. Notice that trajectory Tr_1 is the nearest trajectory to Q_P during *T* but it is not the answer object because Tr_1 does not cross *CR* inside *T*. If the query duration *T* is changed to $[t_2, t_4]$, the result set S_{rslt} will be changed to $\{Tr_4, Tr_5\}$ as well. Also notice that trajectory Tr_6, the nearest trajectory to Q_T inside $T = [t_2, t_4]$, is not included in the S_{rslt} because it does not cross *CR* within *T*.



(a) C2NN$_\mathrm{P}$ and C2NN$_\mathrm{T}$ queries



(b) HCC2NN$_\mathrm{P}$ and HCC2NN$_\mathrm{T}$ queries

Fig. 3 Example of C*k*NN$_\mathrm{P}$, C*k*NN$_\mathrm{T}$, HCC*k*NN$_\mathrm{P}$, and HCC*k*NN$_\mathrm{T}$ queries on moving object trajectories for *k*=2

Definition 3 (HCC*k*NN query over moving object trajectories) *Given D, q* (either Q_P or Q_T)*, T, CR, and k, a historical continuous constrained k-nearest neighbor (HCCkNN) query with respect to q, denoted as HCCkNN*$_P$ *and HCCkNN*$_T$ *respectively, returns from D* at any time instance *of T, the k moving object trajectories that are closest to q and cross* (*or completely fall into*) *CR. The answer set contains k lists* l_i ($1 \le i \le k$)*, with each* l_i *consisting of tuple* ⟨*Tr*, [t_j, t_k)⟩ *that denotes the trajectory Tr is the i-th NN of q during* [t_j, t_k) ⊆ *T.*

For instance, an example *HCC2NN*$_P$ query is issued at Q_P, with $D = \{Tr_1, Tr_2, Tr_3, Tr_4, Tr_5, Tr_6, Tr_7\}$, $T = [t_1, t_4]$, and *CR* set to the shadowed area, as shown in Fig. 3(b). As k=2, its result set contains two lists l_1 and l_2, with $l_1 = \{\langle Tr_3, [t_1, t_2)\rangle, \langle Tr_2, [t_2, t_3)\rangle, \langle Tr_3, [t_3, t_4]\rangle\}$, and $l_2 = \{\langle Tr_2, [t_1, t_2)\rangle, \langle Tr_3, [t_2, t_3)\rangle, \langle Tr_4, [t_3, t_4]\rangle\}$. It indicates that trajectories Tr_3 and Tr_2 are the top-2 NNs to Q_P during $[t_1, t_2)$, trajectories Tr_2 and Tr_3 are the top-2 NNs to Q_P during $[t_2, t_3)$, and so on. Suppose another *HCC2NN*$_T$ query is issued at a trajectory Q_T and $T = [t_2, t_5]$. The result contains $l_1 = \{\langle Tr_6, [t_2, t_3)\rangle, \langle Tr_7, [t_3, t_4)\rangle, \langle Tr_6, [t_4, t_5]\rangle\}$, and $l_2 = \{\langle Tr_5, [t_2, t_3)\rangle, \langle Tr_6, [t_3, t_4)\rangle, \langle Tr_7, [t_4, t_5]\rangle\}$.

4 Algorithms for C*k*NN$_P$ queries

C*k*NN queries naturally involve both range queries and *k*NN searches. Consequently, a straightforward approach, namely C*k*NN$_P$-SR, is to call *k*NN search algorithm to report the trajectories according to ascending orders of their distances to the query point. For each reported trajectory *Tr*, it will be included in the answer set if and only if *Tr* intersects *CR*. The process continues until there are *k* trajectories contained in the answer set or it is for sure that the rest of the trajectories do not intersect *CR*. Alternatively, we can first conduct a range query to retrieve all the candidate trajectories that are within *CR* and then invoke *k*NN retrieval to find the *k* nearest ones, namely C*k*NN$_P$-RS. Although both approaches can return the right answer set for C*k*NN$_P$ retrieval, neither one is efficient. In this section, we present two algorithms for handling the C*k*NN$_P$ query on moving object trajectories that can seamlessly integrate the range search and NN traversal together to improve the search performance, namely, *CkNN*$_P$*-DF algorithm* and *CkNN*$_P$*-BF algorithm*.

4.1 C*k*NN$_P$-DF algorithm

C*k*NN$_P$-DF provides the ability to process C*k*NN search with respect to Q_P during *T*, as shown in Algorithm 1. In fact, C*k*NN$_P$-DF adapts the Point*k*NNSearch algorithm proposed in [11] by merging the region constraint into the algorithm. The details of the C*k*NN$_P$-DF algorithm are as follows.

The result set *kNearest* maintains tuples ⟨*E*, *d*⟩ such that *E* is one of the *k*NN objects to Q_P known so far and its distance to Q_P is *d*. Parameter *kNearest.MaxDist* stores the maximum of *d* stored in *kNearest*, which is initialized to be infinity (line 1). At the leaf level of the tree structure that indexes trajectory data, C*k*NN$_P$-DF iteratively accesses each leaf entry *E* in the leaf node *N* (lines 2–6). In particular, C*k*NN$_P$-DF invokes an algorithm GetEntryInConstraint (depicted in Algorithm 2) to check whether *E* intersects (or is enclosed by) *CR* during *T* (line 4). If so, the GetEntryInConstraint interpolates *E* to produce *E'* (i.e., a portion of *E*) whose temporal component is within *T* as well as spatial component is contained in *CR*, and returns TRUE; otherwise, it returns FALSE to indicate that *E* for sure does not contribute to the result set. It is noticed that if GetEntryInConstraint returns

TRUE, CkNN$_P$-DF calculates the Euclidean distance between E' and Q_P during T (i.e., $MinDist\ (E', Q_P, T)$), includes $\langle E', MinDist\ (E', Q_P, T)\rangle$ to *kNearest* if $MinDist\ (E', Q_P, T) <$ *kNearest.MaxDist* holds, and updates *kNearest.MaxDist* if necessary (lines 5–6). At the non-leaf level of the tree structure, CkNN$_P$-DF recursively visits every child entry of the intermediate (i.e., non-leaf) node (lines 7–12). When a potential candidate is retrieved, the algorithm, backtracking to the upper level, prunes the nodes in the active branch list (line 12) using the pruning heuristics proposed in [6, 29]. Note that when CkNN$_P$-DF invokes a function GenBranchList to generate a node's branch list (line 8), we also combine region constraint into the GenBranchList. For this purpose, an algorithm GetNodeInConstraint (presented in Algorithm 3) is applied.

Algorithm 1 Depth-First based CkNN$_P$ query algorithm (CkNN$_P$-DF)

Input: N: a node of the TB-tree (initially is the root); Q_P; T; CR

Output: *kNearest*: the structure storing the final query result

```
 1: initialize $kNearest = \emptyset$ and $kNearest.MaxDist = \infty$
 2: if $N$ is a leaf node then
 3:    for each leaf entry $E \in N$ do
 4:       if GetEntryInConstraint ($E, E', T, CR$) then
 5:          if $MinDist\ (E', Q_P, T) < kNearest.MaxDist$ then
 6:             add $E'$ with $MinDist\ (E', Q_P, T)$ to $kNearest$ and update $kNearest.MaxDist$ if necessary
 7: else   // $N$ is an intermediate (i.e., non-leaf) node
 8:    $BranchList$ = GenBranchList ($N, Q_P, T, CR$)
 9:    SortBranchList ($BranchList$)   // sort active branch list by $MinDist$ metric
10:    for each child node entry $E$ in $BranchList$ do
11:       CkNN_P-DF ($E, Q_P, T, CR, kNearest$)
12:       PruneBranchList ($BranchList$)   // use pruning heuristics in [6, 29] to prune $BranchList$
```

Function GenBranchList (N, Q_P, T, CR)

```
 1: for each entry $E$ in $N$ do
 2:    if GetNodeInConstraint ($E, E', T, CR$) then
 3:       add $E'$ to branch list $list$ together with its $MinDist\ (E', Q_P, T)$
 4: return $list$
```

The GetEntryInConstraint algorithm, as depicted in Algorithm 2, is to refine a given trajectory segment such that its spatial component is contained in CR and its time extent is within T. The algorithm proceeds as follows. Initially, GetEntryInConstraint checks whether the time period of trajectory segment TS overlaps T (line 1). If not (see Fig. 4(a)), the algorithm is terminated by returning FALSE since TS does not satisfy the specified time condition; otherwise, a linear interpolation is applied in order to compute TS's portion located inside T, denoted by *ConstraintTS* (line 3). Next, GetEntryInConstraint proceeds to determine whether the *spatial bound* of *ConstraintTS*, i.e., *ConstraintTS.spatialbound*, overlaps CR. Here, we distinguish the following three cases: (i) If *ConstraintTS. spatialbound* does not overlap CR (see Fig. 4(b)), then GetEntryInConstraint returns FALSE and gets terminated (lines 4–5). (ii) If *ConstraintTS.spatialbound* is completely bounded by CR (see Fig. 4(c)), then GetEntryInConstraint returns TRUE and is stopped (lines 6–7). (iii) If *ConstraintTS.spatialbound* overlaps CR and there exists at least one intersection between a boundary *bdy* of CR and *ConstraintTS* (see Fig. 4(d)), GetEntryInConstraint computes the intersection and then interpolates *ConstraintTS* to produce the portion whose spatial extent is included in CR completely (lines 8–15). In Fig. 4(d) the thick solid line is the qualifying portion of *ConstraintTS*, whereas the thick dashed line is the pruned portion of *ConstraintTS*, as it falls out of *CR*.

Algorithm 2 Get entries falling in restricted area algorithm (GetEntryInConstraint)

```
Input:   TS: a trajectory segment; ConstraintTS: the trajectory segment that is enclosed by CR; T; CR
Output:  Boolean value TRUE together with a part of TS whose spatial component is contained in CR
         and time extent is within T if TS crosses (or fully falls into) CR; otherwise, return FALSE

 1: if ($TS.t_s$, $TS.t_e$) does not overlap ($T.t_s$, $T.t_e$) then
 2:    return FALSE
    // interpolate TS to produce ConstraintTS within T
 3: ConstraintTS = Interpolate (TS, Max ($TS.t_s$, $T.t_s$), Min ($TS.t_e$, $T.t_e$))
 4: if ConstraintTS.spatialbound does not overlap CR then
 5:    return FALSE
 6: else if CR includes ConstraintTS.spatialbound then
 7:    return TRUE    // ConstraintTS's spatial bound is enclosed by CR
 8: IntersectionFlag = FALSE
 9: for each boundary bdy of CR do
10:    if bdy intersects ConstraintTS then
11:       IntersectionFlag = TRUE
12:       compute the intersection p between bdy and ConstraintTS
13:       interpolate ConstraintTS to get the portion whose spatial extent falls into CR using p
14:    else
15:       continue    // for the next for-loop
16: return IntersectionFlag
```

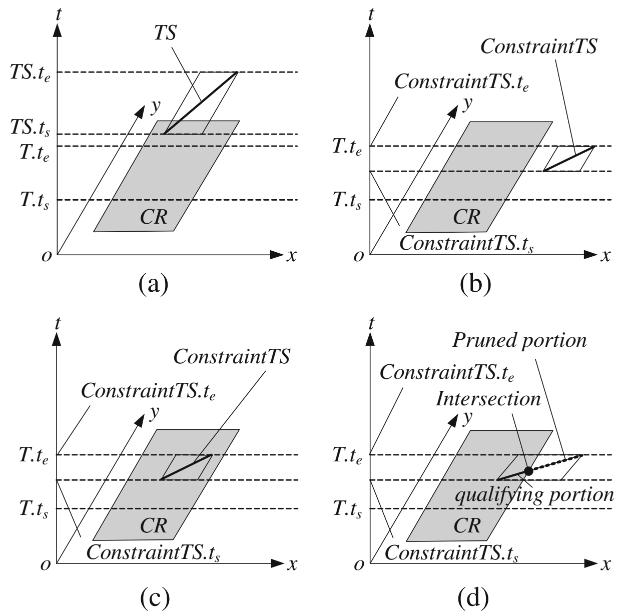


Fig. 4 Illustration of GetEntryInConstraint algorithm

Similar to GetEntryInConstraint, the GetNodeInConstraint algorithm can verify whether the time interval of a given intermediate node entry *N* overlaps *T*, and if *N*'s spatial extent intersects (or is contained in) *CR*, with its pseudo-code outlined in Algorithm 3. If yes, GetNodeInConstraint interpolates *N* to produce *ConstraintN*, the portion of *N* whose temporal extent is within *T* and spatial area is enclosed by *CR*, and returns TRUE; otherwise, it returns FALSE.

Algorithm 3 Get nodes falling in restricted area algorithm (GetNodeInConstraint)

Input: *N*: a tree node; *ConstraintN*: the node that is enclosed by *CR*; *T*; *CR*

Output: Boolean value TRUE together with a part of *N* whose spatial component is contained in *CR* and time extent is within *T* if *N* crosses (or fully falls into) *CR*; otherwise, return FALSE

```
 1: if ($N.t_s$, $N.t_e$) does not overlap ($T.t_s$, $T.t_e$) then
 2:     return FALSE
    // interpolate N to produce ConstraintN within T
 3: ConstraintN = Interpolate (N, Max ($N.t_s$, $T.t_s$), Min ($N.t_e$, $T.t_e$))
 4: for each dimension dim in spatial dimensions of N do
 5:     if ConstraintN overlaps CR then
 6:         interpolate ConstraintN to get the portion overlapping the dim
 7:     else
 8:         return FALSE
 9: return TRUE
```

4.2 C*k*NN$_P$-BF algorithm

C*k*NN$_P$-BF follows the *best-first* traversal paradigm and enables effective pruning strategies to discard all unnecessary entries. Algorithm 4 shows the pseudo-code of C*k*NN$_P$-BF algorithm.

Algorithm 4 Best-First based C*k*NN$_P$ query algorithm (C*k*NN$_P$-BF)

Input: *R*: a TB-tree built on the set of moving object trajectories; Q_P; *T*; *CR*; *k*

Output: S_{rslt}: the set of the *k* trajectories that lie closest to Q_P and cross (or fully fall into) *CR* during *T*

```
 1: initialize heap hp = ∅ and insert all entries of the root in R into hp
 2: while hp is not empty do
 3:     de-heap the top entry E from hp
 4:     if E is an actual trajectory segment entry and its identifier id is not in $S_{rslt}$ then
 5:         insert E as an answer object into $S_{rslt}$ if $|S_{rslt}| < k$; otherwise, return $S_{rslt}$
 6:     else if E is a leaf node then
 7:         MinimalDist = ∞
 8:         for each entry $e \in E$ do
 9:             if GetEntryInConstraint (e, e', T, CR) and MinDist (e', $Q_P$, T) < MinimalDist then
10:                 MinimalDist = MinDist (e', $Q_P$, T) and NearestE = e'
11:         insert NearestE with MinimalDist into hp
12:     else    // E is an intermediate (i.e., non-leaf) node
13:         for each entry $e \in E$ do
14:             if GetNodeInConstraint (e, e', T, CR) then
15:                 insert e' with MinDist (e', $Q_P$, T) into hp
16: return $S_{rslt}$
```

Starting from the root node of the tree R on the set of trajectory data, CkNN$_P$-BF recursively traverses R in the *best-first* fashion (lines 2–15). More specifically, CkNN$_P$-BF first de-heaps the top entry E from the heap hp (line 3). If E is an actual entry of trajectory segment and its identifier id is not included in the current query result set S_{rslt}, it is added as an answer trajectory to S_{rslt} provided that the number of moving object trajectories in S_{rslt} is smaller than k (line 5). If the size of the result set reaches k after this insertion, the algorithm can be terminated as all the answer trajectories have been found. When E is a node entry, there are two possible cases, explained as follows: (i) If E is a leaf node, then CkNN$_P$-BF chooses the entry in E that has the smallest distance to Q_P within T, denoted by *NearestE*, and inserts it into hp (lines 7–11). Here, we also employ our proposed pruning heuristics in [12] to prune away the non-qualifying entries that can not contribute to the query result, in order to reduce the number of node accesses and facilitate the execution of the algorithm. (ii) If E is a non-leaf node, then CkNN$_P$-BF visits only the qualifying nodes that may contain the actual answer trajectories, and inserts them into hp (lines 13–15). Note that CkNN$_P$-BF first invokes the GetEntryInConstraint (GetNodeInConstraint) algorithm to determine if every entry e in a node E satisfies both the temporal and spatial constraints in line 9 (line 14) before visiting e. This checking is necessary because it can filter out those entries in E that do not meet the given constraints.

5 Algorithms for CkNN$_T$ queries

In this section, we extend our approaches to tackle the CkNN$_T$ retrieval over moving object trajectories. The CkNN$_T$ search is a variation of the CkNN$_P$ query with the following difference: a CkNN$_T$ query is issued at a query trajectory instead of a query point. Corresponding to our previously proposed CkNN$_P$-SR, CkNN$_P$-RS, CkNN$_P$-DF, and CkNN$_P$-BF algorithms for CkNN$_P$ queries, we develop CkNN$_T$-SR, CkNN$_T$-RS, CkNN$_T$-DF, and CkNN$_T$-BF algorithms respectively, to answer CkNN$_T$ retrieval. In the following, we omit the descriptions of CkNN$_T$-RS and CkNN$_T$-SR algorithms as they are straightforward, but only present the details of CkNN$_T$-DF and CkNN$_T$-BF algorithms.

5.1 CkNN$_T$-DF algorithm

CkNN$_T$-DF can handle the CkNN retrieval with respect to a given query trajectory, following a *depth-first* manner. It shares the same principle as CkNN$_P$-DF algorithm, with its pseudo-code listed in Algorithm 5. Initially, a linear interpolation is called to get the actual query trajectory Q_T' having the time interval across T (line 2). Subsequently, CkNN$_T$-DF traverses the tree R on D in a *depth-first* fashion (lines 3–17). In particular, at the leaf level, CkNN$_T$-DF computes the smallest Euclidean distance between two trajectory segments during T (line 10). Similar to CkNN$_P$-DF, for every leaf entry E in the leaf node N, CkNN$_T$-DF first invokes the GetEntryInConstraint algorithm (discussed in Section 4.1) to examine whether E satisfies the time constraints and the spatial constraints (line 5). In addition, for every query trajectory segment TS in Q_T', CkNN$_T$-DF needs invoke interpolate to produce a pair of entry nE and trajectory segment TS' with the identical temporal extent, before computing the distance from *TS* to the leaf entry accessed currently (lines 8–9).

Algorithm 5 Depth-First based CkNN$_T$ query algorithm (CkNN$_T$-DF)

```
Input:   N: a node of the TB-tree (initially is the root); Q_T; T; CR
Output:  kNearest: the structure storing the final query result

 1: initialize kNearest = ∅ and kNearest.MaxDist = ∞
    // get the actual query trajectory having the interval across T
 2: Q_T' = Interpolate (Q_T, Max (Q_T.t_s, T.t_s), Min (Q_T.t_e, T.t_e))
 3: if N is a leaf node then
 4:    for each leaf entry E ∈ N do
 5:       if GetEntryInConstraint (E, E', T, CR) then
 6:          for each trajectory segment entry TS ∈ Q_T' do
 7:             if (TS.t_s, TS.t_e) overlaps (E'.t_s, E'.t_e) then
 8:                nE = Interpolate (E', Max (E'.t_s, TS.t_s), Min (E'.t_e, TS.t_e))
 9:                TS' = Interpolate (TS, Max (E'.t_s, TS.t_s), Min (E'.t_e, TS.t_e))
10:                if MinDist (TS', nE, T) < kNearest.MaxDist then
11:                   add nE with MinDist (TS', nE, T) to kNearest and update
                      the value of kNearest.MaxDist if necessary
12: else    // N is an intermediate (i.e., non-leaf) node
13:    BranchList = GenTrajectoryBranchList (N, Q_T', T, CR)
14:    SortBranchList (BranchList)
15:    for each child node entry E in BranchList do
16:       CkNN_T-DF (E, Q_T', T, CR, kNearest)
17:       PruneBranchList (BranchList)

Function GenTrajectoryBranchList (N, Q_T, T, CR)
 1: for each entry E ∈ N do
 2:    if GetNodeInConstraint (E, E', T, CR) then
 3:       Q_T' = Interpolate (Q_T, Max (Q_T.t_s, E'.t_s), Min (Q_T.t_e, E'.t_e))
 4:       add E' to branch list list together with its Mindist_Trajectory_Rectangle (Q_T', E')
 5: return list
```

At a non-leaf level of *R*, CkNN$_T$-DF recursively visits every child entry *E* of the intermediate node *N* (lines 13–17). Nevertheless, unlike CkNN$_P$-DF, CkNN$_T$-DF employs the GenTrajectoryBranchList function instead of the GenBranchList function to generate the node's branch list with the entries satisfying the given constraints (line 13). Like GenBranchList, GenTrajectoryBranchList also calls the GetNodeInConstraint algorithm to check if *E* in *N* meets the specified constraints before it expands *E*. Moreover, in GenTrajectoryBranchList we utilize the *Mindist_Trajectory_Rectangle* metric developed in [11] to calculate the minimal distance between the query trajectory and the MBB of *N* (line 4 of the GenTrajectoryBranchList function). It must be pointed out that we do not need to compute *Mindist_Trajectory_Rectangle* against Q_T, but only against the part Q_T' of Q_T being inside the temporal extent of the *N*'s MBB by interpolating.

Algorithm 6 Best-First based C*k*NN$_T$ query algorithm (C*k*NN$_T$-BF)

Input: *R*: a TB-tree on the set of moving object trajectories; Q_T; *T*; *CR*; *k*

Output: S_{rslt}: the set of the *k* trajectories that lie closest to Q_T and cross (or fully fall into) *CR* during *T*

```
 1: initialize heap $hp = \varnothing$ and insert all entries of the root in $R$ into $hp$
 2: get the set $S_{QT}$ of the actual query trajectory segments having the time extents across $T$
 3: while $hp$ is not empty do
 4:     de-heap the top entry $E$ from $hp$
 5:     if $E$ is an actual trajectory segment entry and its identifier $id$ is not in $S_{rslt}$ then
 6:         insert $E$ as an answer object into $S_{rslt}$ if $|S_{rslt}| < k$; otherwise, return $S_{rslt}$
 7:     else if $E$ is a leaf node then
 8:         $MinimalDist = \infty$
 9:         for each entry $e \in E$ do
10:             if GetEntryInConstraint ($e, e', T, CR$) then
11:                 for each trajectory segment entry $TS \in S_{QT}$ do
12:                     if ($TS.t_s, TS.t_e$) overlaps ($e'.t_s, e'.t_e$) then
13:                         $ne$ = Interpolate ($e'$, Max($e'.t_s, TS.t_s$), Min($e'.t_e, TS.t_e$))
14:                         $TS'$ = Interpolate ($TS$, Max($e'.t_s, TS.t_s$), Min($e'.t_e, TS.t_e$))
15:                         if $MinDist(TS', ne, T) < MinimalDist$ then
16:                             $MinimalDist = MinDist(TS', ne, T)$ and $NearestE = ne$
17:         insert $NearestE$ into $hp$ together with its $MinimalDist$
18:     else    // $E$ is an intermediate (i.e., non-leaf) node
19:         for each entry $e \in E$ do
20:             if GetNodeInConstraint ($e, e', T, CR$) then
21:                 insert $e'$ into $hp$ along with $Mindist\_Trajectory\_Rectangle(Q_T, e')$
22: return $S_{rslt}$
```

5.2 C*k*NN$_T$-BF algorithm

Following a *best-first* traversal paradigm, C*k*NN$_T$-BF can deal with the C*k*NN retrieval with respect to the predefined query trajectory. The C*k*NN$_T$-BF algorithm is illustrated in Algorithm 6. Initially, C*k*NN$_T$-BF initializes a heap *hp*, inserts all the entries in the root node of the tree *R* on *D* into *hp*, and obtains the actual query trajectory whose time interval overlaps *T* (lines 1–2). Then, C*k*NN$_T$-BF iterates the following operations until it finds the final query result (lines 3–21). Specifically, it first de-heaps the top entry *E* from *hp* (line 4). If *E* is an actual entry of trajectory segment and it is not contained in S_{rslt}, C*k*NN$_T$-BF inserts *E* as an answer object into S_{rslt} if the cardinality of S_{rslt} (i.e., $|S_{rslt}|$) is less than *k*; otherwise, it returns S_{rslt} to terminate the search (lines 5–6). If *E* is a leaf node, only the entry in *E* that has the minimal distance to the actual query trajectory inside *T*, denoted as

NearestE, is inserted into *hp* (lines 8–17). On the other hand, if *E* is an intermediate node, C*k*NN$_T$-BF retrieves only the qualified nodes that can contribute to the final query result and en-heaps the *hp* (lines 19–21). Like C*k*NN$_P$-BF, C*k*NN$_T$-BF first calls the GetEntryInConstraint (GetNodeInConstraint) algorithm to examine whether each entry *E* in a node *N* satisfies both the temporal and spatial constraints in line 10 (line 20) in order to avoid any unnecessary visiting. The checking is required as some entries in *E* may not meet the specified constraints (therefore they do not need to be visited).

6 Algorithms for HCC*k*NN$_P$ queries

In this section, we describe the algorithms for processing the historical continuous C*k*NN retrieval with respect to stationary query point, i.e., HCC*k*NN$_P$ search. Section 6.1 and Section 6.2 present depth-first based HCC*k*NN$_P$ (called HCC*k*NN$_P$-DF) query algorithm and best-first based HCC*k*NN$_P$ (called HCC*k*NN$_P$- BF) query algorithm, respectively. We omit the discussion of the two-step algorithms for HCC*k*NN$_P$ queries, including (i) NN search followed by a range query for HCC*k*NN$_P$ (called HCC*k*NN$_P$-SR) and (ii) range query followed by NN search for HCC*k*NN$_P$ (called HCC*k*NN$_P$-RS).

Algorithm 7 Depth-First based HCC*k*NN$_P$ query algorithm (HCC*k*NN$_P$-DF)

```
Input:   N: a node of the TB-tree (initially is the root); $Q_P$; T; CR
Output:  kNearestLists: the k nearest lists that store the final query result
1:  initialize kNearestLists.MaxDist = $\infty$
2:  if N is a leaf node then
3:      for each leaf entry $E \in N$ do
4:          if GetEntryInConstraint (E, E', T, CR) then
5:              MDist = ConstructMovingDistance ($Q_P$, E')
6:              if MDist.Dmin < kNearestLists.MaxDist then
7:                  UpdatekNearests (MDist, kNearestLists)    // algorithm of [13]
8:  else    // N is an intermediate (i.e., non-leaf) node
9:      BranchList = GenBranchList (N, $Q_P$, T, CR)
10:     SortBranchList (BranchList)
11:     PruneHContBranchList (BranchList, kNearestLists, kNearestLists.MaxDist)
12:     for each child node entry E in BranchList do
13:         HCCkNN$_P$-DF (E, $Q_P$, T, CR, kNearestLists)
            // Prune all entries having MinDist greater than kNearestLists.MaxDist in BranchList
14:         PruneHContBranchList (BranchList, kNearestLists, kNearestLists.MaxDist)
```

6.1 HCC*k*NN$_P$-DF algorithm

HCC*k*NN$_P$-DF deals with the HCC*k*NN$_P$ retrieval in a depth-first manner. Algorithm 7 depicts the pseudo-code of HCC*k*NN$_P$-DF algorithm. It utilizes a structure *MDist* (line 5), which retains the parameters of the distance function, the associated minimum *Dmin* and maximum *Dmax* of the distance function during the lifetime, a time period, and the actual entry in order to report it as the actual answer object instantly. The structure is calculated based on the *ConstructMovingDistance* function presented in [11]. It needs to note that in the line 7 of Algorithm 7, the structure *kNearestLists*, i.e., the *k* nearest lists storing the final query result, is introduced. Let *kNearestLists.MaxDist* be the maximum of all distances stored inside *kNearestLists*. Then, *kNearestLists.MaxDist* (which is initialized to ∞) can be employed as a pruning threshold to prune those unnecessary entries and branches at the non-leaf level. In particular, any entry having its smallest distance to Q_P within *T* greater than *kNearestLists.MaxDist* can be discarded immediately. Also notice that, our previously proposed Update*k*Nearests algorithm in [13] is employed to update *kNearestLists* structure efficiently. Please refer to [13] for more details.

At the leaf level of the tree structure that indexes trajectory data, HCC*k*NN$_P$-DF invokes GetEntryInConstraint to examine whether every leaf entry *E* in the leaf node *N* accessed currently crosses (or falls into completely) *CR* inside *T* (line 4), before the algorithm visits *E*. This examination is necessary, since the final query result must satisfy the specified constraints. At a non-leaf level of the tree structure, HCC*k*NN$_P$-DF recursively visits each child entry of the intermediate node (lines 8–14). When a potential candidate is retrieved, the algorithm, backtracking to the upper level, prunes the node entries in the active branch list (line 11) using the following pruning heuristics: HCC*k*NN$_P$-DF first compares the *MinDist* of every entry *N* in *BranchList* (i.e., the minimal distance from *N* to Q_P inside *T*) with *kNearestLists.MaxDist*; and then, it computes the largest distance in the *kNearestLists* structure during the time extent of *N*. Next, the algorithm discards all entries having *MinDist* greater than the one computed.

6.2 HCC*k*NN$_P$-BF algorithm

Employing the BF traversal paradigm, HCC*k*NN$_P$-BF processes the HCC*k*NN retrieval with respect to a predefined static query point. To achieve this target, it maintains a heap storing all candidate entries together with their smallest distances to the given query point within *T* (i.e., *MinDist*); these distances are sorted in ascending order of their *MinDist*. The HCC*k*NN$_P$-BF algorithm is shown in Algorithm 8.

Starting from the root node in the tree *R* on *D*, it traverses recursively the tree in a *best-first* fashion (lines 2–17). Specifically, HCC*k*NN$_P$-BF first de-heaps the top entry *E* from *hp* (line 3). If $E.Dmin \geq PruneDist(k)$, that is, the smallest distance between *E* and Q_P inside *T* is not smaller than the maximal distance stored among the *k*-th nearest list, then it reports *kNearestLists* as the final result and terminates (line 5), because the distances from the remaining entries in *hp* to Q_P during *T* are all larger than or equal to *PruneDist*(*k*). In fact, lines 4–5 prevent the non-qualifying entries that do not contribute to the query result from en-heaping there. Next, the algorithm considers the following cases: (i) If *E* is an actual entry of trajectory segment, then HCC*k*NN$_P$-BF invokes Update*k*Nearests algorithm to insert *E* into *kNearestLists* and update *kNearestLists* if necessary (line 7). (ii) If *E* is a leaf node, HCC*k*NN$_P$-BF only adds every entry *e* in *E* to *hp* (lines 9–13) if the spatial region of *e* intersects (or is contained in) *CR* within *T* (using the GetEntryInConstraint algorithm) and *e*'s smallest distance from Q_P inside *T* is smaller than *PruneDist*(*k*). (iii) If *E* is a non-leaf node, HCC*k*NN$_P$-BF also only en-heaps each child entry *e* in *E* (lines 15–17) if *e*'s spatial area crosses (or fully falls into) *CR* within *T* (using the GetNodeInConstraint algorithm) and *e*'s minimal distance to Q_P during *T* is smaller than *PruneDist*(*k*).

Algorithm 8 Best-First based HCC*k*NN$_P$ query algorithm (HCC*k*NN$_P$-BF)

Input: *R*: a TB-tree built on the set of moving object trajectories; Q_P; *T*; *CR*; *k*
Output: *kNearestLists*: the *k* nearest lists that store the final query result

```
1:  initialize heap $hp = \varnothing$ and add all entries of the root in $R$ to $hp$, lists $kNearestLists$ and $PruneDist$
2:  while $hp$ is not empty do
3:      de-heap the top entry $E$ from $hp$
4:      if $E.Dmin \geq PruneDist(k)$ then
5:          return $kNearestLists$   // report the final query result
6:      if $E$ is an actual trajectory segment entry then
7:          UpdatekNearests ($E$, $kNearestLists$)   // update $kNearestLists$
8:      else if $E$ is a leaf node then
9:          for each entry $e \in E$ do
10:             if GetEntryInConstraint ($e$, $e'$, $T$, $CR$) then
11:                 $MDist = ConstructMovingDistance(Q_P, e')$
12:                 if $MDist.Dmin < PruneDist(k)$ then
13:                     insert $e'$ into $hp$ together with its $MDist$
14:     else   // $E$ is an intermediate (i.e., a non-leaf) node
15:         for each entry $e \in E$ do
16:             if GetNodeInConstraint ($e$, $e'$, $T$, $CR$) and $MinDist(e', Q_P, T) < PruneDist(k)$ then
17:                 insert $e'$ with $MinDist(e', Q_P, T)$ into $hp$
18: return $kNearestLists$
```

7 Algorithms for HCC*k*NN$_T$ queries

In this section, we propose our methods for dealing with the HCC*k*NN$_T$ retrieval on the trajectories of moving objects, where the query object is a moving trajectory instead of a stationary point. Following the common methodology proposed previously, we develop four HCC*k*NN$_T$ query algorithms, termed as HCC*k*NN$_T$-SR, HCC*k*NN$_T$-RS, HCC*k*NN$_T$-DF, and HCC*k*NN$_T$-BF, respectively. Here we only focus on the details of both HCC*k*NN$_T$-DF and HCC*k*NN$_T$-BF algorithms.

7.1 HCC*k*NN$_T$-DF algorithm

HCC*k*NN$_T$-DF follows the depth-first traversal paradigm to solve the HCC*k*NN retrieval with respect to a given query trajectory. Algorithm 9 depicts the HCC*k*NN$_T$-DF algorithm. In general, HCC*k*NN$_T$-DF is similar to HCC*k*NN$_P$-DF, with the following differences: (i) At the leaf level, HCCkNN$_T$-DF utilizes the *ConstructMovingDistance* function to compute the distance between two trajectory segments of moving objects rather than one moving object trajectory segment and one stationary point (line 10). (ii) At a non-leaf level, HCCkNN$_T$-DF uses the GenTrajectoryBranchList function instead of the GenBranchList function to generate the node's branch list with the entries satisfying the given constraints (line 14). Like CkNN$_T$-DF, HCCkNN$_T$-DF employs a linear interpolation to get the actual query trajectory Q_T' having the time interval across T before it starts traversing the tree on the trajectory data in a depth-first manner (line 2). For each leaf entry E in the leaf node N, HCCkNN$_T$-DF first invokes the GetEntryInConstraint algorithm to determine whether or not the temporal interval of E overlaps with T, and if E's spatial extent intersects (or is contained in) CR (line 5). Furthermore, for every query trajectory segment entry TS in Q_T' and before calculating the distance from TS to the leaf entry accessed currently during T, HCCkNN$_T$-DF first interpolates to produce a tuple of *entry—trajectory segment* with identical time extent (lines 8–9).

Algorithm 9 Depth-First based HCCkNN$_T$ query algorithm (HCCkNN$_T$-DF)

Input: N: a node of the TB-tree (initially is the root); Q_T; T; CR
Output: *kNearestLists*: the k nearest lists that store the final query result

```
 1: initialize kNearestLists.MaxDist = $\infty$
    // get the actual query trajectory having the time interval across T
 2: $Q_T'$ = Interpolate ($Q_T$, Max ($Q_T.t_s$, $T.t_s$), Min ($Q_T.t_e$, $T.t_e$))
 3: if N is a leaf node then
 4:   for each leaf entry $E \in N$ do
 5:     if GetEntryInConstraint (E, E', T, CR) then
 6:       for each trajectory segment entry TS in $Q_T'$ do
 7:         if ($TS.t_s$, $TS.t_e$) overlaps ($E'.t_s$, $E'.t_e$) then
 8:           nE = Interpolate (E', Max ($E'.t_s$, $TS.t_s$), Min ($E'.t_e$, $TS.t_e$))
 9:           TS' = Interpolate (TS, Max ($E'.t_s$, $TS.t_s$), Min ($E'.t_e$, $TS.t_e$))
10:           MDist = ConstructMovingDistance (TS', nE)
11:           if MDist.Dmin < kNearest.MaxDist then
12:             UpdatekNearests (MDist, kNearestLists)
13: else   // N is an intermediate (i.e., non-leaf) node
14:   BranchList = GenTrajectoryBranchList (N, $Q_T'$, T, CR)
15:   SortBranchList (BranchList)
16:   PruneHContBranchList (BranchList, kNearestLists, kNearestLists.MaxDist)
17:   for each child node entry E in BranchList do
18:     HCCkNN$_T$-DF (E, $Q_T'$, T, CR, kNearestLists)
19:     PruneHContBranchList (BranchList, kNearestLists, kNearestLists.MaxDist)
```

7.2 HCCkNN$_T$-BF algorithm

By adopting the best-first traversal paradigm, HCCkNN$_T$-BF aims at processing the HCkNN search with respect to a specified query trajectory. The HCCkNN$_T$-BF algorithm is shown in Algorithm 10. As with HCCkNN$_P$-BF (cf. Section 6.2), HCCkNN$_T$-BF implements an ordered *best-first* traversal, by starting with the root node of the tree R on D and proceeding down the tree.

Algorithm 10 Best-First based HCCkNN$_T$ query algorithm (HCCkNN$_T$-BF)

Input: R: a TB-tree built on the set of moving object trajectories; Q_T; T; CR; k
Output: *kNearestLists*: the k nearest lists that store the final query result

```
1:  initialize heap $hp = \varnothing$ and add all entries of the root in $R$ to $hp$, lists $kNearestLists$ and $PruneDist$
2:  get the set $S_{QT}$ of actual query trajectory segments having the time periods across $T$
3:  while $hp$ is not empty do
4:      de-heap the top entry $E$ from $hp$
5:      if $E.Dmin \geq PruneDist(k)$ then
6:          return $kNearestLists$    // report the final $k$ nearest lists
7:      if $E$ is an actual trajectory segment entry then
8:          Update$k$Nearests ($E$, $kNearestLists$)
9:      else if $E$ is a leaf node then
10:         for each entry $e \in E$ do
11:             if GetEntryInConstraint ($e$, $e'$, $T$, $CR$) then
12:                 for each trajectory segment entry $TS \in S_{QT}$ do
13:                     if ($TS.t_s$, $TS.t_e$) overlaps ($e'.t_s$, $e'.t_e$) then
14:                         $ne$ = Interpolate ($e'$, Max ($e'.t_s$, $TS.t_s$), Min ($e'.t_e$, $TS.t_e$))
15:                         $TS'$ = Interpolate ($TS$, Max ($e'.t_s$, $TS.t_s$), Min ($e'.t_e$, $TS.t_e$))
16:                         $MDist = ConstructMovingDistance(TS', ne)$
17:                         if $MDist.Dmin < PruneDist(k)$ then
18:                             Insert $ne$ into $hp$ together with its $MDist$
19:     else    // $E$ is an intermediate (i.e., non-leaf) node
20:         for each entry $e \in E$ do
21:             if GetNodeInConstraint ($e$, $e'$, $T$, $CR$) then
22:                 for each trajectory segment entry $TS \in S_{QT}$ do
23:                     if ($TS.t_s$, $TS.t_e$) overlaps ($e'.t_s$, $e'.t_e$) then
24:                         $TS'$ = Interpolate ($TS$, Max ($e'.t_s$, $TS.t_s$), Min ($e'.t_e$, $TS.t_e$))
25:                         if $Mindist\_Trajectory\_Rectangle(TS', e') < PruneDist(k)$ then
26:                             insert $e'$ into $hp$ together with its $Mindist\_Trajectory\_Rectangle(TS', e')$
27: return $kNearestLists$
```

In the first place, HCCkNN$_T$-BF initializes some auxiliary structures including heap *hp*, lists *kNearestLists* and *PruneDist*, inserts all the entries in the root of R into the heap *hp*, and obtains the set S_{QT} of actual query trajectory segments whose time intervals overlap with T (line 1). Subsequently, HCCkNN$_T$-BF recursively finds the answer trajectory that is stored in *kNearestLists* (lines 3–26). In each iteration, HCCkNN$_T$-BF first de-heaps the top entry E from *hp* (line 4). Like HCCkNN$_P$-BF, if $E.Dmin \geq PruneDist(k)$ holds, then HCCkNN$_T$-BF returns *kNearestLists* and terminates since the final result has been discovered (line 6). Otherwise, the algorithm deals with either an actual entry of trajectory segment (lines 7–8) or a node entry containing a leaf node one (lines 9–18) and a non-leaf node one (lines 19–26). Specifically, (i) if E is a trajectory segment entry, then HCCkNN$_T$-BF calls Update$k$Nearests algorithm to add E to *kNearestLists* and update *kNearestLists* (if necessary); (ii) if E is a leaf node, then HCCkNN$_T$-BF inserts for every entry e in E into *hp* if e has the time period across T, e's time interval overlaps with that of each entry TS in S_{QT}, and its distance from TS is smaller than $PruneDist(k)$; similarly, (iii) HCCkNN$_T$-BF adds all the qualifying entries in E to *hp* when E is a non-leaf node. It is important to note that the operation concerned in line 13 is necessary because the temporal extent of some TS in S_{QT}

may not intersect that of e in E (hence it needs not be accessed). Also notice that, the computation of the *Mindist_Trajectory_Rectangle* metric involved in line 25 uses the approach presented in [11].

8 Performance evaluation

In this section, we evaluate the efficiency and scalability of our proposed algorithms in terms of the I/O (i.e., number of node/page accesses) and CPU cost, via extensive experiments on both real and synthetic datasets. All algorithms used in experiments (that are listed in Table 2) were coded in Visual Basic and run on a PC with Pentium IV 3.0 GHz CPU and 1 GB RAM. Note that since two-step algorithms (i.e., CkNN$_P$-RS, CkNN$_T$-RS, HCCkNN$_P$-RS, and HCCkNN$_T$-RS) are always worse than the single-step algorithms by several orders of magnitude, they are omitted in our presented experimental results. However, it is worth noting that CkNN$_P$-RS, CkNN$_T$-RS, HCCkNN$_P$-RS, and HCCkNN$_T$-RS are good choices when the given constrained region CR is *extremely small*.

8.1 Experimental setup

We use two real datasets that consist of a fleet of trucks containing 276 trajectories and a fleet of school buses containing 145 trajectories. Both of them are from the *R-tree Portal*[2]. We also deploy several synthetic datasets generated by a GSTD data generator [37] to examine the scalability of the algorithms. Specifically, the synthetic data correspond to 100, 200, 400, 800, and 1600 moving objects, with the position of each object being sampled approximately 1500 times. Furthermore, the initial distribution of moving objects follows *Gaussian* distribution while their movement follows random distribution. Table 3 summarizes the statistics of both real and synthetic datasets.

Each dataset is indexed by a TB-tree [27], using a page size of 4 K bytes and a buffer having capacity varying from 10% of the tree size to 1000 pages. Five factors, including CR, k, time duration (T), the number of moving objects ($\#MO$), and buffer size (bs), that may affect the performance of the algorithms are evaluated. The parameter values used in our experiments are presented in Table 4. In each set of experiments, 100 queries are issued and the average performance of last 50 queries is measured, with the first 50 queries warming up the buffer. In addition, the query points Q_P are randomly generated in the 2D space. Similarly, the query trajectories Q_T are generated randomly as well. In particular, when we evaluate both CkNN$_T$ and HCCkNN$_T$ queries on trucks dataset, we take random trajectory segments from the school buses dataset as Q_T; while on GSTD datasets, the query trajectories are also created by the GSTD data generator.

8.2 Results on CkNN$_P$ query algorithm

The first set of experiments studies the impact of CR on CkNN$_P$ performance. The size of CR varies from 10% to 70% of the whole data space. Notice that even the smallest CR in the experiments has a reasonable selectivity, that is, it covers more than k number of moving object trajectories with k specified by the query. Figure 5 illustrates the number of node accesses and CPU time (in seconds) of the algorithms as a function of CR by using the trucks and GSTD datasets and fixing k=4 (i.e., a median value used in Fig. 6) and T=6% (i.e., a

[2] The URL of the R-tree Portal is http://www.rtreeportal.org/.

Table 2 All algorithms used in experiments

Query type	Algorithm	Description
The C*k*NN retrieval w.r.t. Q_P (i.e., C*k*NN$_\mathrm{P}$)	C*k*NN$_\mathrm{P}$-SR	NN search followed by a Range query for C*k*NN$_\mathrm{P}$
	C*k*NN$_\mathrm{P}$-RS	Range query followed by NN search for C*k*NN$_\mathrm{P}$
	C*k*NN$_\mathrm{P}$-DF	Depth-First based C*k*NN$_\mathrm{P}$ retrieval
	C*k*NN$_\mathrm{P}$-BF	Best-First based C*k*NN$_\mathrm{P}$ retrieval
The C*k*NN retrieval w.r.t. Q_T (i.e., C*k*NN$_\mathrm{T}$)	C*k*NN$_\mathrm{T}$-SR	NN search followed by a Range query for C*k*NN$_\mathrm{T}$
	C*k*NN$_\mathrm{T}$-RS	Range query followed by NN search for C*k*NN$_\mathrm{T}$
	C*k*NN$_\mathrm{T}$-DF	Depth-First based C*k*NN$_\mathrm{T}$ retrieval
	C*k*NN$_\mathrm{T}$-BF	Best-First based C*k*NN$_\mathrm{T}$ retrieval
The HCC*k*NN retrieval w.r.t. Q_P (i.e., HCC*k*NN$_\mathrm{P}$)	HCC*k*NN$_\mathrm{P}$-SR	NN search followed by a Range query for HCC*k*NN$_\mathrm{P}$
	HCC*k*NN$_\mathrm{P}$-RS	Range query followed by NN search for HCC*k*NN$_\mathrm{P}$
	HCC*k*NN$_\mathrm{P}$-DF	Depth-First based HCC*k*NN$_\mathrm{P}$ retrieval
	HCC*k*NN$_\mathrm{P}$-BF	Best-First based HCC*k*NN$_\mathrm{P}$ retrieval
The HCC*k*NN retrieval w.r.t. Q_T (i.e., HCC*k*NN$_\mathrm{T}$)	HCC*k*NN$_\mathrm{T}$-SR	NN search followed by a Range query for HCC*k*NN$_\mathrm{T}$
	HCC*k*NN$_\mathrm{T}$-RS	Range query followed by NN search for HCC*k*NN$_\mathrm{T}$
	HCC*k*NN$_\mathrm{T}$-DF	Depth-First based HCC*k*NN$_\mathrm{T}$ retrieval
	HCC*k*NN$_\mathrm{T}$-BF	Best-First based HCC*k*NN$_\mathrm{T}$ retrieval

median value used in Fig. 7). Clearly, C*k*NN$_\mathrm{P}$-BF consistently outperforms the other algorithms. As the *CR* becomes smaller, the selectivity of the *CR* grows, and hence the advantage of the C*k*NN$_\mathrm{P}$-BF algorithm over other algorithms becomes more significant. For instance, when *CR*=10%, the C*k*NN$_\mathrm{P}$-BF only requires 5.6% CPU time of C*k*NN$_\mathrm{P}$-SR algorithm, but when *CR*=70%, it requires 41.8% CPU time of C*k*NN$_\mathrm{P}$-SR algorithm. In addition, we observe that C*k*NN$_\mathrm{P}$-DF performs better than C*k*NN$_\mathrm{P}$-SR when the size of *CR* is small (e.g., 20%), but it loses its advantage when *CR* becomes larger (e.g., 60%). The reason behind is that the number of unnecessary node accesses incurred by the C*k*NN$_\mathrm{P}$-DF retrieval increases with *CR*.

For the rest of this section, the *CR* value is set to its default values, i.e., 20% and 60%. Furthermore, we only present the performance of I/O cost because the performance of CPU cost shares the same trend as that of I/O cost.

In the second set of experiments, we evaluate the influence of *k*, the number of required answer trajectories, on the performance of different algorithms. The result is depicted in Fig. 6, with the value of *k* varying from 1 to 16. In all the cases, C*k*NN$_\mathrm{P}$-BF exceeds the

Table 3 Statistics of real and synthetic datasets

Datasets	No. trajectories	No. entries	No. pages
Trucks	276	112,203	835
School buses	145	66,096	466
GSTD 100	100	150,052	1,008
GSTD 200	200	300,101	2,015
GSTD 400	400	600,203	4,029
GSTD 800	800	1,200,430	8,057
GSTD 1,600	1,600	2,400,889	16,112

Table 4 Parameters in experiments

Parameters	Description	Values	Default values
k	number of nearest neighbors	1, 2, 4, 8, 16	4
CR (% of full space)	constrained region	10, 20, 30, 40, 50, 60, 70	10, 20, 40, 60
T (% of entire interval)	temporal extent	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	3, 6
#*MO*	number of moving objects	100, 200, 400, 800, 1600	400
bs (% of the tree size)	buffer size	0, 5, 10, 15, 20	10% to 1000 pages

other algorithms by several orders of magnitude, and the difference becomes more significant as k increases. In addition, we observe that when CR is small (e.g., 20%), CkNN$_{\mathrm{P}}$-DF is more effective than CkNN$_{\mathrm{P}}$-SR. However, CkNN$_{\mathrm{P}}$-SR becomes more competitive when CR becomes large (e.g., 60%), as it only applies constraint checks to candidate trajectories returned by previous NN search on D.

The next set of experiments explores the effect of T on the performance of different algorithm, with the result plotted in Fig. 7. Again, CkNN$_{\mathrm{P}}$-BF performs best in all the cases. The I/O cost of all algorithms increases slightly as T grows, which is caused by the increase of temporal overlapping. As we can see from the Fig. 7, a small value of T may be more expensive than a larger one. This irregularity is due to the positions and the temporal extents of the query trajectories.

Figure 8 measures the I/O cost of the algorithms as a function of #*MO* by using GSTD data sets and fixing k=4 and T=6%. Again, CkNN$_{\mathrm{P}}$-BF evidently outperforms the other methods. Moreover, the performance difference increases with the growth of the dataset size.

As mentioned in Section 8.1, all the above experiments are conducted with an LRU buffer whose size is from 10% of the tree size to 1000 pages. In the last set of experiments, we inspect

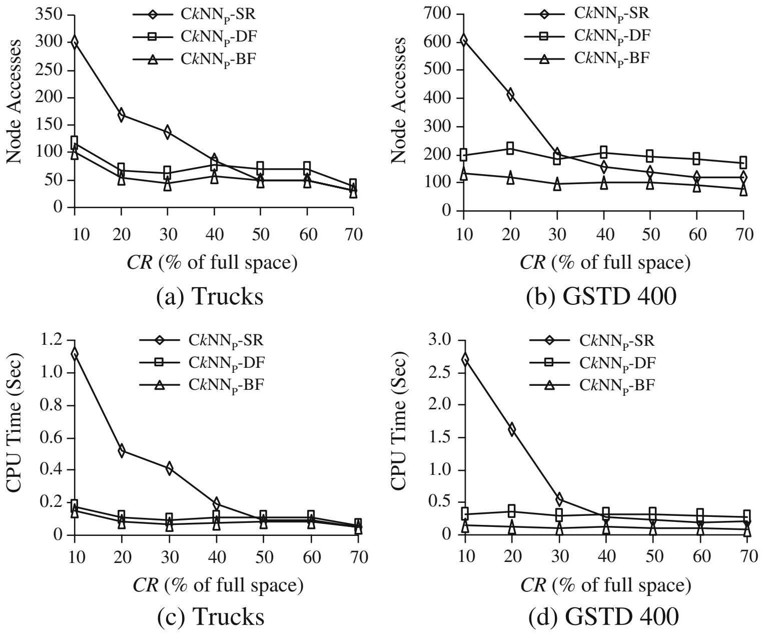


Fig. 5 I/O cost and CPU time (sec) vs. CR (k=4, T=6%)

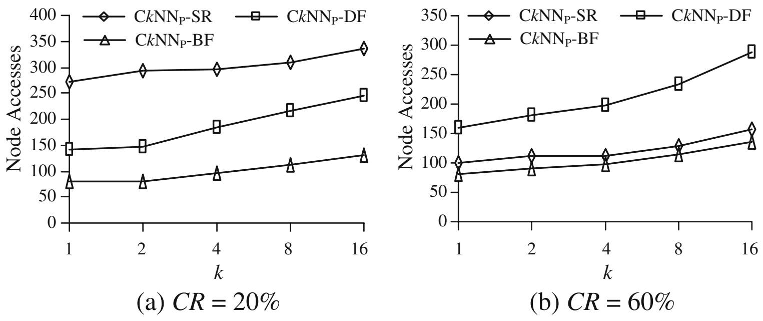


Fig. 6 I/O cost vs. *k* (*T*=6%, GSTD 400)

the impact of *bs* using GSTD data sets. Towards this, we fix *k* and *T* to their default values (i.e., 4 and 6% respectively) and vary the buffer size from 0% to 20% of the TB-tree size. To obtain stable statistics, we measure the average cost of the last 50 queries, after the first 50 queries have been performed for warming up the buffer, as mentioned earlier. Figure 9 plots the overall query cost (i.e., the sum of the I/O time and CPU time, where the I/O time is computed by charging 10 ms for each page access) with respect to the buffer size. Note that C_P-SR, C_P-DF, and C_P-BF shown on the top of each bar represent C*k*NN_P-SR, C*k*NN_P-DF, and C*k*NN_P-BF, respectively. As the buffer size increases, the I/O cost drops, but the CPU cost remains almost the same. C*k*NN_P-BF again outperforms its competitors in all cases.

8.3 Results on C*k*NN_T query algorithm

Having examined the efficiency and scalability of the algorithms for C*k*NN_P queries, we proceed to evaluate the performance of the algorithms for C*k*NN_T queries. In Fig. 10, we present the influence of *CR* on the real and synthetic datasets. Similar to the charts shown in Fig. 5, C*k*NN_T-BF performs consistently the best in all the cases. Both C*k*NN_T-DF and C*k*NN_T-SR are affected significantly by the size of *CR*. In the rest of this section, we fix *CR* size to 10% and 40%, respectively.

The impacts of *k*, *T*, and #*MO* on the I/O cost are presented in Figs. 11, 12, and 13 respectively, and the effect of *bs* on total query cost is depicted in Fig. 14, where C_T-SR, C_T-DF, and C_T-BF shown on the top of each bar denote C*k*NN_T-SR, C*k*NN_T-DF, and C*k*NN_T-BF, respectively. All the algorithms perform similarly, compared against C*k*NN_P query as illustrated in Figs. 6, 7, 8, and 9.

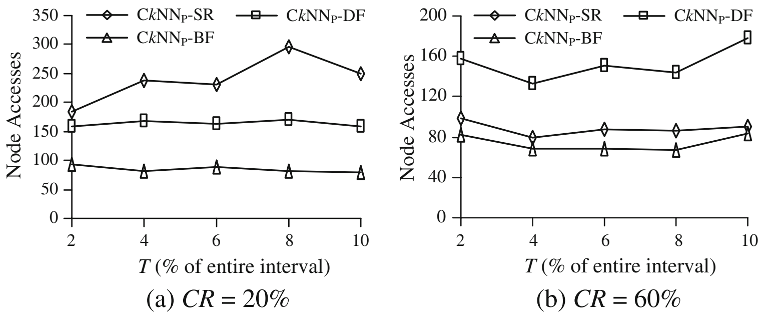


Fig. 7 I/O cost vs. *T* (*k*=4, GSTD 400)

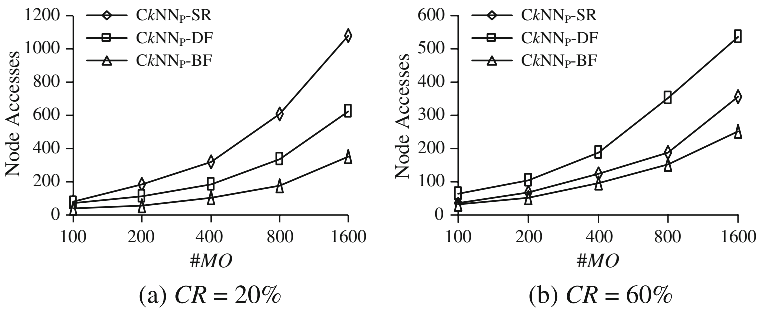


Fig. 8 I/O cost vs. *#MO* (*k*=4, *T*=6%)

8.4 Results on HCC*k*NN$_P$ query algorithm

Next, we are going to evaluate the efficiency of different algorithms for HCC*k*NN$_P$ queries. In this section, we first evaluate the effect of *CR* on the performance of the algorithms, fixing *k*=4 and *T*=3%. It is noticed that the temporal extend (i.e., *T*) varies from 1% to 5%, which is different from that under C*k*NN queries. This is because compared with C*k*NN, processing of HCC*k*NN retrieval is much more expensive. The longer the *T* is, the more the overhead is. In order to reduce the simulation overhead, we only evaluate the performance with small *T*. Figure 15 plots the cost of the algorithms as a function of *CR* by varying *CR* from 10% to 70% of the entire data space. As expected, HCC*k*NN$_P$-BF performs the best in all the cases. It demonstrates a stable performance as *CR* changes, while both HCC*k*NN$_P$-SR and HCC*k*NN$_P$-DF are sensitive to the size of *CR*. When we compare HCC*k*NN$_P$-SR and HCC*k*NN$_P$-DF in terms of I/O cost, HCC*k*NN$_P$-DF performs better when *CR* is small (e.g., *CR*=20%); whereas it loses its advantage when larger *CR*s are encountered (e.g., *CR*= 60%), as seen from Fig. 15(a) and (b). For CPU cost, HCC*k*NN$_P$-SR outperforms HCC*k*NN$_P$-DF in most cases, especially for large values of *CR*. In addition, the performance difference between HCC*k*NN$_P$-BF and HCC*k*NN$_P$-SR is almost negligible when *CR* is greater than a certain value (e.g., *CR*=50% in Fig. 15(a)).

In subsequent experiments, we fix the *CR* value to 20% and 60% respectively, and compare the performance of the algorithms on both real and synthetic datasets with respect to the other parameters, including *k*, *T*, *#MO*, and *bs*.

The second set of experiments inspects the impact of *k*. Setting *T*=3%, Fig. 16 examines the cost of alternative methods by altering *k* from 1 to 16. HCC*k*NN$_P$-BF outperforms the

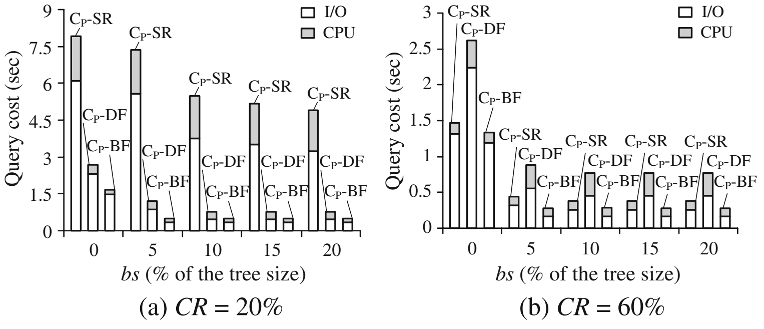


Fig. 9 Query cost (sec) vs. *bs* (*k*=4, *T*=6%, GSTD 400)

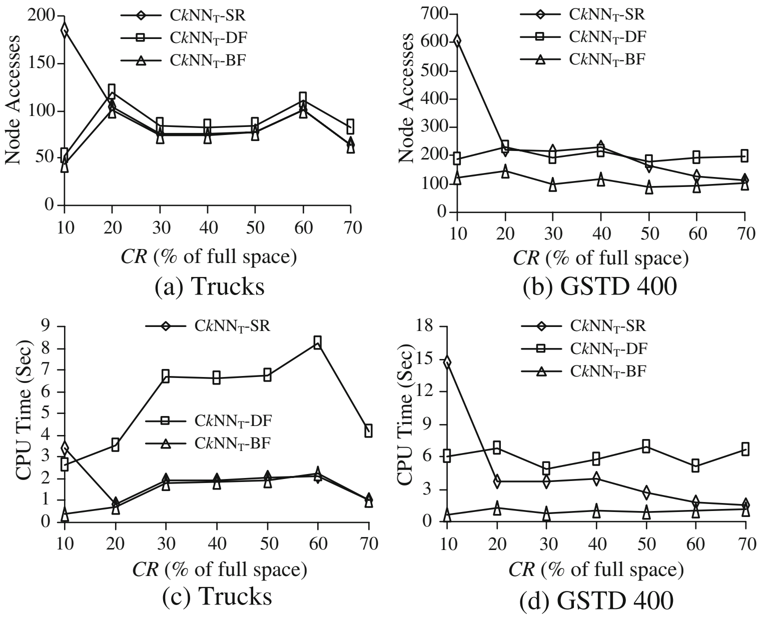


Fig. 10 I/O cost and CPU time (sec) vs. *CR* (k=4, T=6%)

other algorithms in all the cases. From Fig. 16(a) and (b), we can see that when *CR*=20%, the I/O performance of HCCkNN$_P$-DF is obviously better than that of HCCkNN$_P$-SR; but the latter exceeds the former significantly, when *CR*=60%. This phenomenon is also demonstrated in the subsequent experiments of this section. On the other hand, as shown in Fig. 16(c) and (d), HCCkNN$_P$-SR is clearly over HCCkNN$_P$-DF in most cases in terms of CPU cost, although it is still worse than HCCkNN$_P$-BF. This reason is that HCCkNN$_P$-DF costs many unnecessary node accesses and requires more cost for maintaining and updating the k nearest lists (i.e., *kNearestLists* structure).

Next, we study the influence of T on the efficiency of the algorithms. Towards this, we set k to 4, vary T from 1% to 5% of the full space, and illustrate the experimental results in Fig. 17. Again, HCCkNN$_P$-BF is always the best approach in all the cases. Both the I/O cost and the CPU cost of all the algorithms increases with T, which is mainly caused by the growth of temporal overlapping.

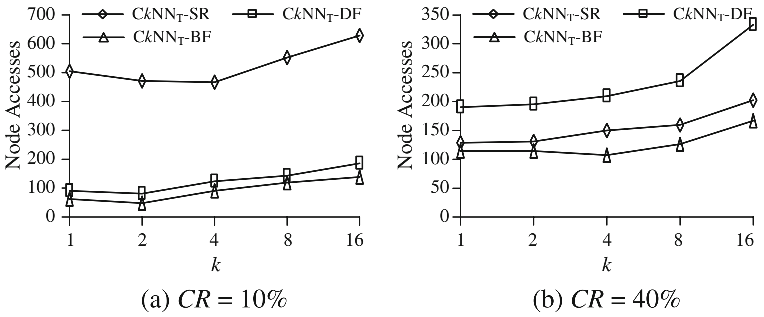


Fig. 11 I/O cost vs. k (T=6%, GSTD 400)

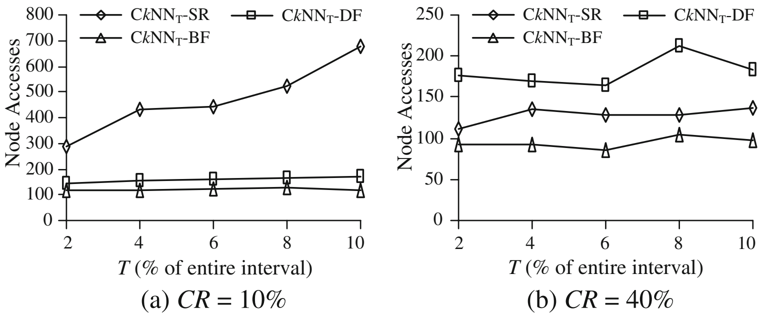


Fig. 12 I/O cost vs. *T* (*k*=4, GSTD 400)

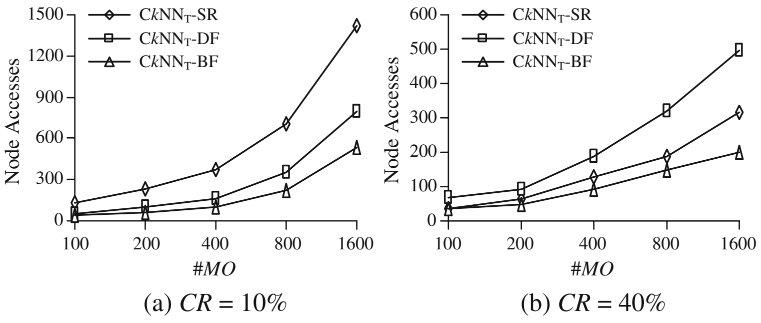


Fig. 13 I/O cost vs. #*MO* (*k*=4, *T*=6%)

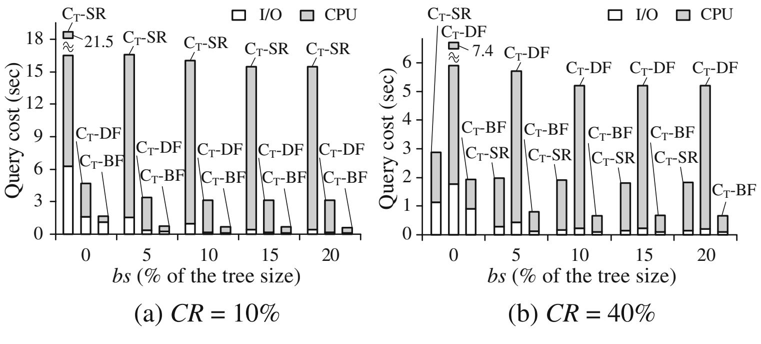


Fig. 14 Query cost (sec) vs. *bs* (*k*=4, *T*=6%, GSTD 400)

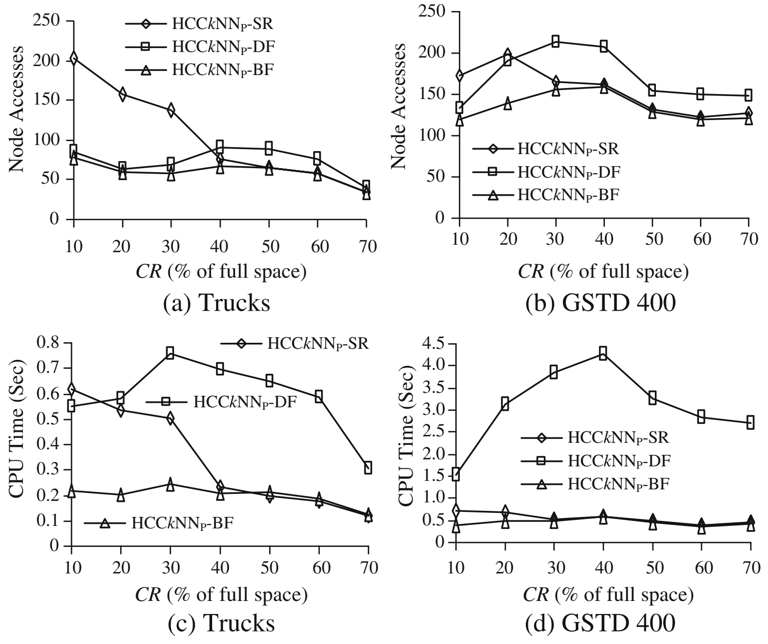


Fig. 15 I/O cost and CPU time (sec) vs. *CR* (k=4, T=3%)

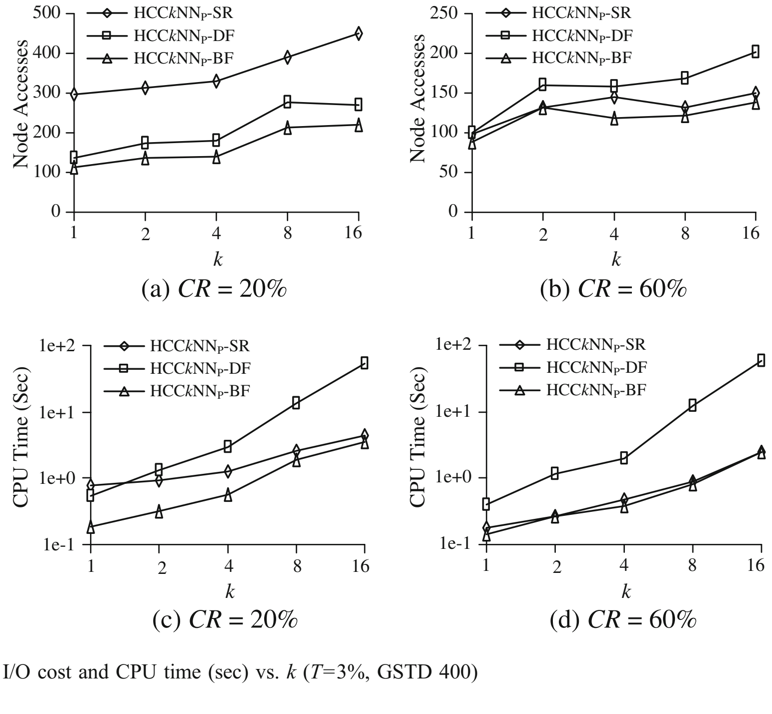


Fig. 16 I/O cost and CPU time (sec) vs. k (T=3%, GSTD 400)

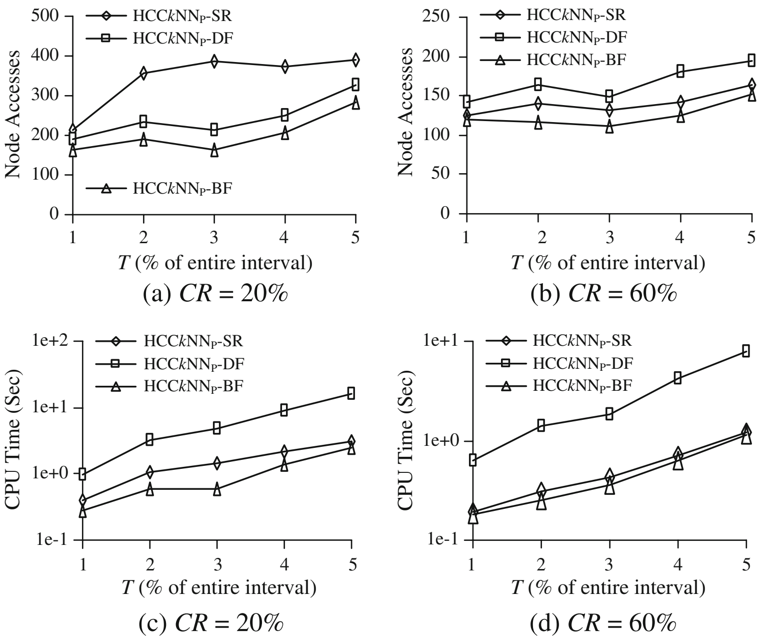


Fig. 17 I/O cost and CPU time (sec) vs. T (k=4, GSTD 400)

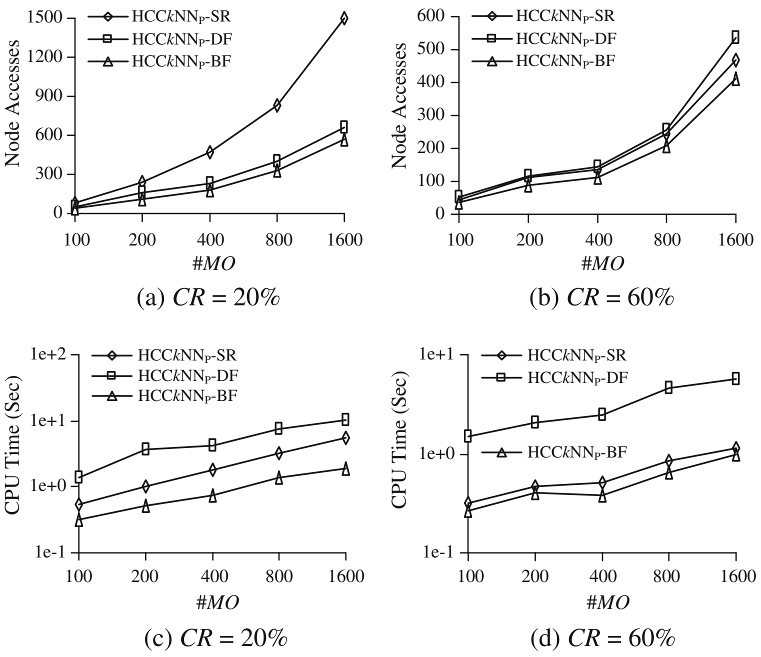


Fig. 18 I/O cost and CPU time (sec) vs. #MO (k=4, T=3%)

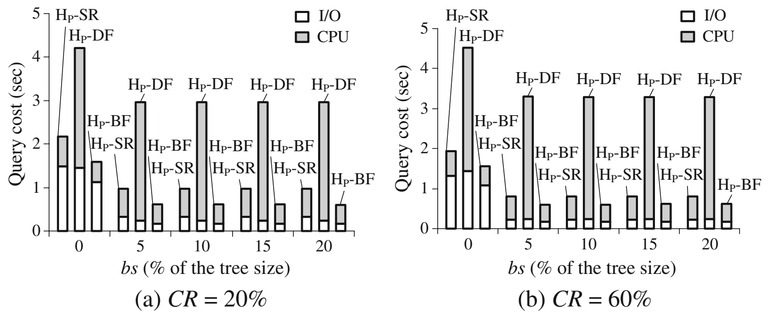


Fig. 19 Query cost (sec) vs. *bs* (*k*=4, *T*=3%, GSTD 400)

The fourth set of experiment in this section evaluates the effect of #*MO* on the performance of the algorithms, as shown in Fig. 18. Also, HCC*k*NN$_P$-BF outperforms the other methods significantly, especially in terms of CPU overhead.

Finally, we investigate the influence of LRU buffers on our proposed algorithms. Similar to the settings of Fig. 9, we fix *k*=4 and *T*=3%, change buffer size (i.e., *bs*) from 0% to 20% of the TB-tree size. Figure 19 illustrates the overall query overhead as a function of *bs*, where H$_P$-SR, H$_P$-DF, and H$_P$-BF shown on the top of each bar stand for HCC*k*NN$_P$-SR, HCC*k*NN$_P$-DF, and HCC*k*NN$_P$-BF, respectively, demonstrating phenomena similar to those in Fig. 9. Specifically, with the growth of the buffer size, the I/O cost decreases, but the CPU cost remains almost the same. Furthermore, for all settings, HCC*k*NN$_P$-BF is consistently better than the other algorithms regardless of the buffer size.

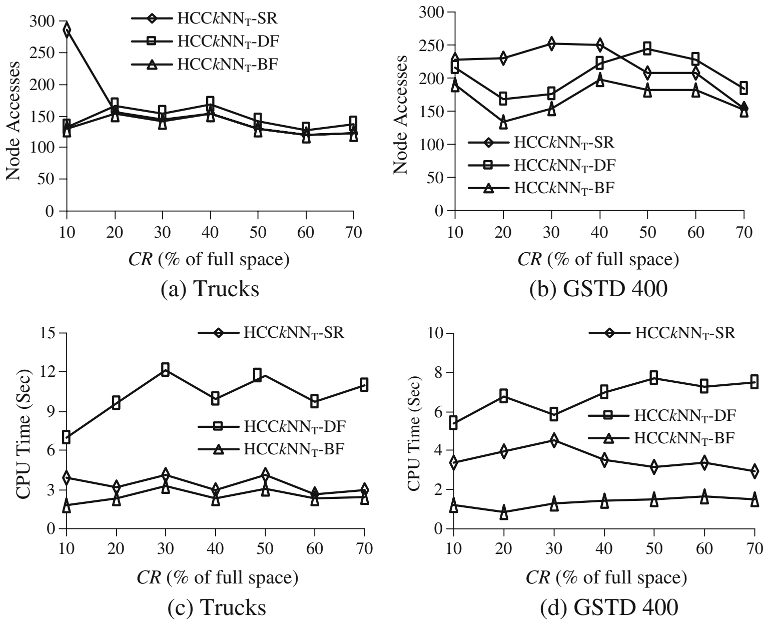


Fig. 20 I/O cost and CPU time (sec) vs. *CR* (*k*=4, *T*=3%)

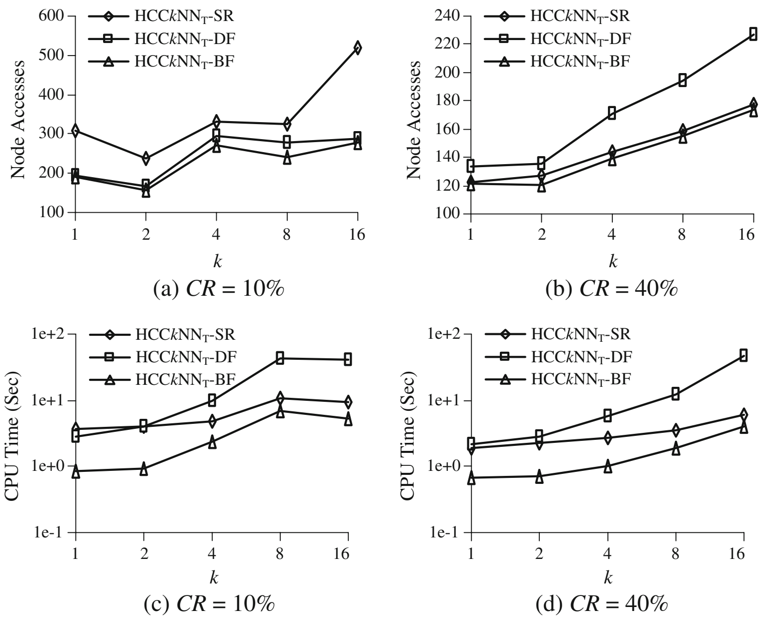


Fig. 21 I/O cost and CPU time (sec) vs. *k* (*T*=3%, GSTD 400)

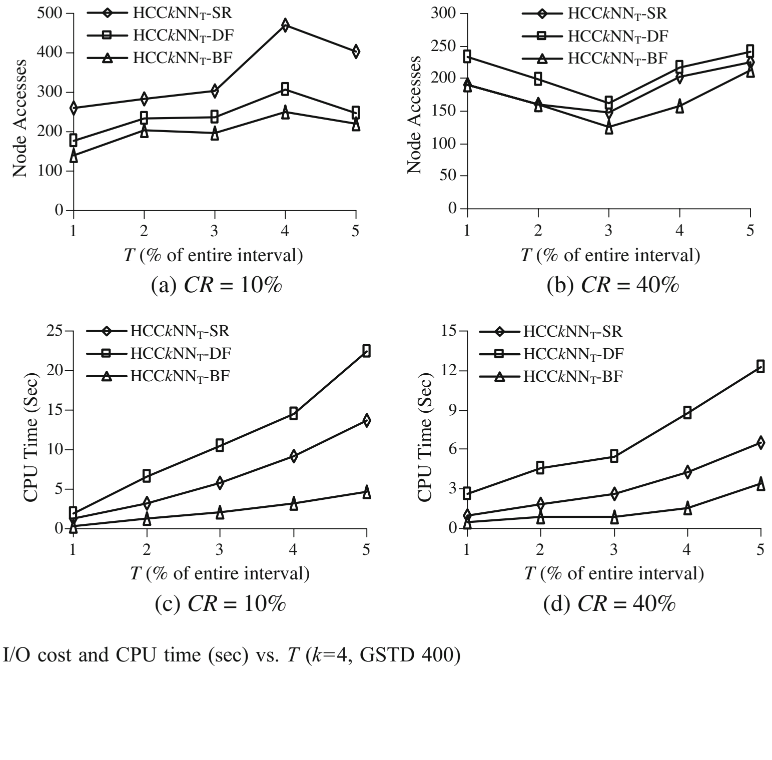


Fig. 22 I/O cost and CPU time (sec) vs. *T* (*k*=4, GSTD 400)

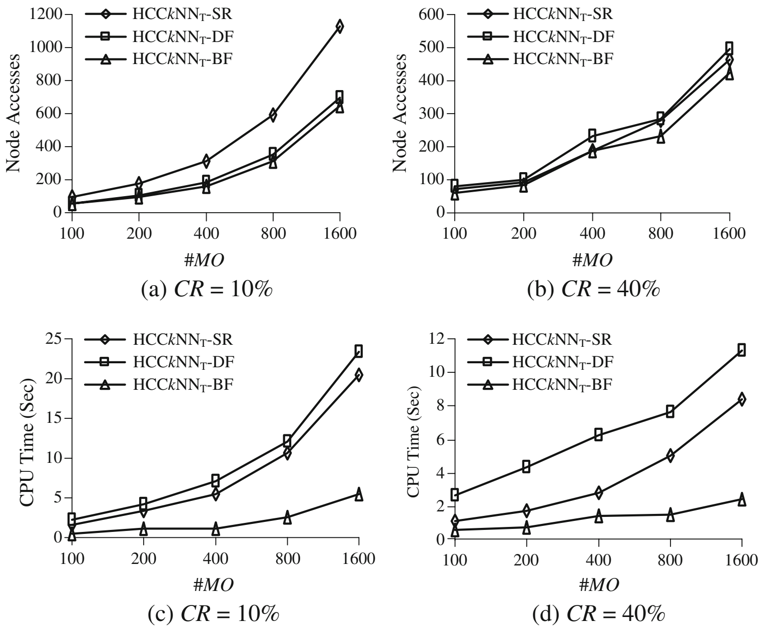


Fig. 23 I/O cost and CPU time (sec) vs. #*MO* (*k*=4, *T*=3%)

8.5 Results on HCCkNN$_T$ query algorithm

Finally, we examine the performance of the algorithms for HCCkNN$_T$ queries. First, the impact of *CR* on the performance is evaluated, with the results depicted in Fig. 20. Obviously, HCCkNN$_T$-BF outperforms the others in all the cases. With respect to I/O cost, HCCkNN$_T$-DF is better than HCCkNN$_T$-SR when small sizes of *CR*s are encountered; and then, the latter outperforms the former as *CR* grows. For CPU cost, even though HCCkNN$_T$-SR is still worse than HCCkNN$_T$-BF, it is clearly more effective than HCCkNN$_T$-DF under all the cases. In the rest of this section, we fix the *CR* size to 10% and 40%, respectively. The efficiency with respect to *k*, *T*, #*MO*, and *bs* are shown in Figs. 21, 22, 23, and 24, respectively. All the observations are consistent with those from

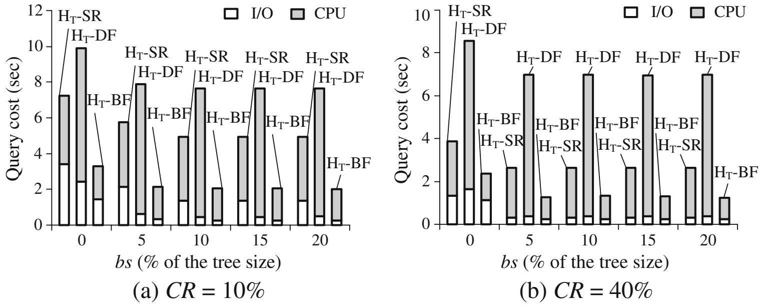


Fig. 24 Query cost (sec) vs. *bs* (*k*=4, *T*=3%, GSTD 400)

previous experiments. Also note that H_T-SR, H_T-DF, and H_T-BF shown on the top of each bar in Fig. 24 represent HCC*k*NN$_T$-SR, HCC*k*NN$_T$-DF, and HCC*k*NN$_T$-BF, respectively.

9 Conclusions

Although unconstrained *k*-nearest neighbor search for moving object trajectories has been studied recently in the database literature, there is no prior work on the constrained *k*-nearest neighbor retrieval over moving object trajectories. In this paper, we introduce C*k*NN and HCC*k*NN queries and propose a suite of algorithms for efficiently processing such queries with different properties and advantages, based on a member of the R-tree family for trajectory data (i.e., TB-tree) without changing the underlying index structure. In particular, we thoroughly investigate two types of C*k*NN queries, viz. C*k*NN$_P$ and C*k*NN$_T$, which are defined with respect to stationary query points and moving query trajectories, respectively; and two types of HCC*k*NN queries, i.e., HCC*k*NN$_P$ and HCC*k*NN$_T$, which are continuous counterparts of C*k*NN$_P$ and C*k*NN$_T$, respectively. An extensive experimental comparison with both real and synthetic datasets verifies the efficiency and scalability of our proposed algorithms, and confirms that best-first based integrated approaches are very effective for answering C*k*NN and HCC*k*NN queries.

In the future, we plan to explore the applicability of other query algorithms (e.g., *k*-closest pair queries [7]) on moving object trajectories. For example, "*find the k pairs of trajectories that have the k smallest distances among all possible pairs during the predefined time interval* [t_s, t_e]". Recently, Arumugam et al. [1] have investigated the *closest-point-of-approach join* for moving object histories. Further studies along this line are also planned for our subsequent research. Finally, it would be particularly interesting to develop a cost model for estimating the execution time of the constrained *k*NN search over trajectory data. Such model will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even better algorithms.

Acknowledgements We would like to thank Elias Frentzos for his useful feedback on the source-codes of the proposed algorithms in [10, 11]. We also would like to express our gratitude to some anonymous reviewers, for giving valuable and helpful comments to improve the technical quality and presentation of this paper.

References

1. Arumugam S, Jermaine C (2006) Closest-point-of-approach join for moving object histories. in Proc. of ICDE, p. 86
2. Beckmann N, Kriegel H-P, Schneider R, Seeger B. (1990) The R*-tree: An efficient and robust access method for points and rectangles. in Proc. of ACM SIGMOD, pp 322–331
3. Benetis R, Jensen CS, Karciauskas G, Saltenis S (2002) Nearest neighbor and reverse nearest neighbor queries for moving objects. in Proc. of IDEAS, pp 44–53
4. Benetis R, Jensen CS, Karciauskas G, Saltenis S (2006) Nearest and reverse nearest neighbor queries for moving objects. VLDB J 15(3):229–249. doi:10.1007/s00778-005-0166-4
5. Berchtold S, Ertl B, Keim DA, Kriegel H-P, Seidl T (1998) Fast nearest neighbor search in high-dimensional space. in Proc. of ICDE, pp 209–218
6. Cheung KL, Fu AW-C (1998) Enhanced nearest neighbour search on the R-tree. SIGMOD Rec 27 (3):16–21. doi:10.1145/290593.290596
7. Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M (2000) Closest pair queries in spatial databases. in Proc. of ACM SIGMOD, pp 189–200
8. Deng K, Zhou X, Shen H, Xu K, Lin X (2006) Surface *k*-NN query processing. in Proc. of ICDE, p. 78

9. Ferhatosmanoglu H, Stanoi I, Agrawal D, Abbadi A (2001) Constrained nearest neighbor queries. in Proc. of SSTD, pp 257–278
10. Frentzos E, Gratsias K, Pelekis N, Theodoridis Y (2005) Nearest neighbor search on moving object trajectories. in Proc. of SSTD, pp 328–345
11. Frentzos E, Gratsias K, Pelekis N, Theodoridis Y (2007) Algorithms for nearest neighbor search on moving object trajectories. GeoInformatica 11(2):159–193. doi:10.1007/s10707-006-0007-7
12. Gao Y, Li C, Chen G, Chen L, Jiang X, Chen C (2007) Efficient *k*-nearest-neighbor search algorithms for historical moving object trajectories. J Comput Sci Technol 22(2):232–244. doi:10.1007/s11390-007-9030-x
13. Gao Y, Li C, Chen G, Li Q, Chen C (2007) Efficient algorithms for historical continuous *k*NN query processing over moving object trajectories. in Proc. of APWeb/WAIM, pp 188–199
14. Gao Y, Chen G, Li Q, Li C, Chen C (2008 Constrained *k*-nearest neighbor query processing over moving object trajectories. in Proc. of DASFAA, pp 635–643
15. Guttman A (1984) R-trees: A dynamic index structure for spatial searching. in Proc. of ACM SIGMOD, pp 47–57
16. Hjaltason GR, Samet H (1999) Distance browsing in spatial databases. ACM Trans Database Syst 24 (2):265–318. doi:10.1145/320248.320255
17. Iwerks GS, Samet H, Smith K (2003) Continuous *k*-nearest neighbor queries for continuously moving points with updates. in Proc. of VLDB, pp 512–523
18. Kollios G, Gunopoulos D, Tsotras V (1999) On indexing mobile objects. in Proc. of ACM PODS, pp 261–272
19. Korn F, Sidiropoulos N, Faloutsos C, Siegel E, Protopapas Z (1996) Fast nearest neighbor search in medical image databases. in Proc. of VLDB, pp 215–226
20. Manolopoulos Y, Nanopoulos A, Papadopoulos AN, Theodoridis Y (2005) R-trees: Theory and applications. Springer-Verlag
21. Mokbel MF, Ghanem TM, Aref WG (2003) Spatio-temporal access methods. IEEE Data Eng Bull 26 (2):40–49
22. Mouratidis K, Hadjieleftheriou M, Papadias D (2005) Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. in Proc. of ACM SIGMOD, pp 634–645
23. Mouratidis K, Papadias D, Bakiras S, Tao Y (2005) A threshold-based algorithm for continuous monitoring of *k* nearest neighbors. IEEE Trans Knowl Data Eng 17(11):1451–1464. doi:10.1109/TKDE.2005.172
24. Mouratidis K, Yiu M, Papadias D, Mamoulis N (2006) Continuous nearest neighbor monitoring in road networks. in Proc. of VLDB, pp 43–54
25. Papadias D, Shen Q, Tao Y, Mouratidis K (2004) Group nearest neighbor queries. in Proc. of ICDE, pp 301–312
26. Papadopoulos AN, Manolopoulos Y (1997) Performance of nearest neighbor queries in R-trees. in Proc. of ICDT, pp 394–408
27. Pfoser D, Jensen CS, Theodoridis Y (2000) Novel approaches in query processing for moving object trajectories. in Proc. of VLDB, pp 395–406
28. Raptopoulou K, Papadopoulos AN, Manolopoulos Y (2003) Fast nearest neighbor query processing in moving object databases. GeoInformatica 7(2):113–137. doi:10.1023/A:1023403908170
29. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. in Proc. of ACM SIGMOD, pp 71–79
30. Saltenis S, Jensen CS, Leutenegger ST, Lopez MA (2000) Indexing the positions of continuously moving objects. in Proc. of ACM SIGMOD, pp 331–342
31. Seidl T, Kriegel H-P (1998) Optimal multi-step *k*-nearest neighbor search. in Proc. of ACM SIGMOD, pp 154–165
32. Sellis T, Roussopoulos N, Faloutsos C (1987) The R$^+$-tree: A dynamic index for multi-dimensional objects. in Proc. of VLDB, pp 507–518
33. Song Z, Roussopoulos N (2001) K-nearest neighbor search for moving query point. in Proc. of SSTD, pp 79–96
34. Tao Y, Papadias D (2001) The MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. in Proc. of VLDB, pp 431–440
35. Tao Y, Papadias D (2002) Time parameterized queries in spatio-temporal databases. in Proc. of ACM SIGMOD, pp 334–345
36. Tao Y, Papadias D, Shen Q (2002) Continuous nearest neighbor search. in Proc. of VLDB, pp 287–298
37. Theodoridis Y, Silva JRO, Nascimento MA (1999) On the generation of spatiotemporal datasets. in Proc. of SSD, pp 147–164

38. Theodoridis Y, Vazirgiannis M, Sellis TK (1996) Spatio-temporal indexing for large multimedia applications. in Proc. of ICMCS, pp 441–448
39. Xiong X, Mokbel M, Aref WG (2005) SEA-CNN: Scalable processing of continuous *k*-nearest neighbor queries in spatio-temporal databases. in Proc. of ICDE, pp 643–654
40. Yu X, Pu K, Koudas N (2005) Monitoring *k*-nearest neighbor queries over moving objects. in Proc. of ICDE, pp 631–642
41. Zheng B, Lee D (2001) Semantic caching in location-dependent query. in Proc. of SSTD, pp 97–116
42. Zhang J, Mamoulis N, Papadias D, Tao Y (2004) All-nearest-neighbors queries in spatial databases. in Proc. of SSDBM, pp 297–306