The VLDB Journal
DOI 10.1007/s00778-014-0361-2

REGULAR PAPER

Continuous nearest-neighbor queries with location uncertainty

A. Prasad Sistla · Ouri Wolfson · Bo Xu

Received: 15 July 2013 / Revised: 17 May 2014 / Accepted: 20 May 2014

Abstract In this paper, we consider the problem of evaluating the continuous query of finding the k nearest objects with respect to a given point object O_q among a set of n moving point-objects. The query returns a sequence of answer-pairs, namely pairs of the form (I, S) such that I is a time interval and S is the set of objects that are closest to O_q during I. When there is uncertainty associated with the locations of the moving objects, S is the set of all the objects that are possibly the k nearest neighbors. We analyze the lower bound and the upper bound on the maximum number of answer-pairs, for the certain case and the uncertain case, respectively. Then, we consider two different types of algorithms. The first is off-line algorithms that compute a priori all the answer-pairs. The second type is on-line algorithms that at any time return the current answer-pair. We present algorithms for the certain case and the uncertain case, respectively, and analyze their complexity. We experimentally compare different algorithms using a database of 1 million objects derived from real-world GPS traces.

Keywords Moving objects databases · Trajectories · Nearest-neighbor queries · Continuous queries · Uncertainty management

Electronic supplementary material The online version of this article (doi:10.1007/s00778-014-0361-2) contains supplementary material, which is available to authorized users.

A. P. Sistla · O. Wolfson
Department of Computer Science,
University of Illinois at Chicago, Chicago, IL, USA

B. Xu (✉)
HERE, Chicago, IL, USA
e-mail: bo.5.xu@here.com

1 Introduction

1.1 Contributions

A CkNN query is a query that continuously finds the k nearest neighbors with respect to a given moving point-object O_q among a set of n moving point-objects. The query returns a sequence of answer-pairs, namely pairs of the form (I, S) such that I is a time interval and S is the set of k objects that are nearest to O_q during I. CkNN queries see many applications in mobile-computing environments. For example, in a road network, the CkNN query can continuously provide a driver with the locations of the k nearest vehicles from her location. The query enables the driver to be aware of the vehicles that are blocked by a truck, a turn, a blind zone, etc. In a digital battlefield, a vehicle may use the CkNN query to monitor the k nearest hostile (or friendly) vehicles for surveillance (or for support). CkNN queries are also used for clustering data streams [31]. In this case, moving objects are sensor observations rather than physical objects and they do not necessarily move in a geo-space. In general, CkNN queries are used in any database application that continuously ranks answers based on their Euclidean distances to an object.

Now, we discuss various models of the CkNN query.

Motion Model: We discuss two motion models, linear and piecewise linear. In the linear model all the objects, including O_q, start from different points and move with a constant velocity-vector in a multi-dimensional space. This model is applicable when the database only knows the initial location and velocity-vector of each moving object. In the piecewise linear model, the objects change their velocities a finite number of times denoted by m. Here are some scenarios in which the piecewise linear model is useful. *Scenario 1*: The

Published online: 10 June 2014

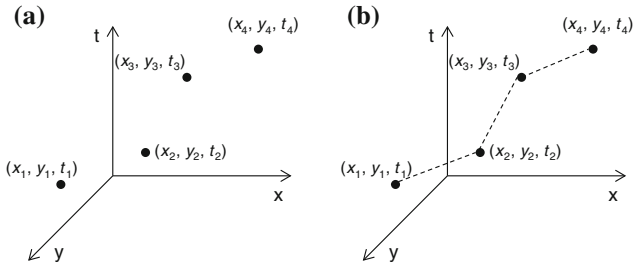

Fig. 1 Scenario 1 for piecewise linear motion. **a** The database is a sequence of historical (location, time) records. **b** The motion between two consecutive records is interpreted as linear

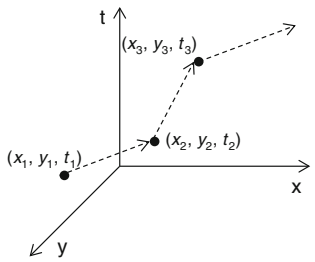

Fig. 2 Scenario 2 for piecewise linear motion: the object updates the database when its velocity-vector changes. Each black point in the figure represents an update

database is given a sequence of historical records for each moving object; each record indicates that an object was at a particular location at a particular time. If we assume that each object o moves linearly between every pair of consecutive records of o, then the motion follows the piecewise linear model (see Fig. 1). *Scenario 2*: Each object sends updates to the database from time to time, where each update includes the object's current location and velocity-vector. The database assumes that the object will move linearly using the current velocity-vector until the next update is received; and the moving object knows that the database makes this assumption, thus it will send an update when the assumption is violated. In this case, the motion of the object in the database follows the piecewise linear model. Observe that this scenario only necessitates that the object knows its current velocity vector; whenever the vector changes, an update is sent to the database (see Fig. 2). *Scenario 3*: The database knows the motion plan of each object such as a delivery truck. A motion plan describes the object's future path in a road network, and a schedule. For example, the object is supposed to arrive at location p_1 at time t_1, at location p_2 at time t_2, etc. Such plans are prepared a priori and stored in the database. Based on the motion plan, the future motion of the object can be described by the piecewise linear model. *Scenario 4*: The piecewise linear model can approximate a nonlinear motion with an arbitrarily high precision. As the number of linear pieces increases, the precision increases.

Other models have been proposed in the literature for representing motion of moving objects. For example, splines are proposed to deal with the situation where continuous first- and second-order derivatives are needed to represent smooth changes of speed and direction (see e.g., [34]). There are also models for representing motion of objects that have an extent (see e.g., [35]). These models may be more accurate and necessary for some applications, but they make query processing more complex.

Certainty model: The certain CkNN query assumes that the location of an object as a function of time is known precisely. In this case, for each answer-pair (I, S), S is exactly the set of k objects that are nearest to O_q during time interval I. However, uncertainty is an inherent aspect in databases that manage locations of moving objects. Due to measurement imprecision (such as GPS errors), continuous motion, and network delays, the location of a moving object stored in the database will not always precisely represent its real location. In some scenarios, location uncertainty is deliberately introduced. For example, in order to protect privacy in location based services, uncertainty is injected to a user's location (see e.g., [22]). Specifically, we consider the uncertainty model in which at any point in time the location of a moving object can be anywhere in a circular uncertainty region, but not outside it (see e.g., [10,13]). Furthermore, we assume that the location of a moving object within its uncertainty region is independent of the locations of the other moving objects within their regions. The reason for this assumption is that often the locations in the database are obtained from localization devices on board the moving objects, and devices of different objects report independently. The motion of different objects is sometimes correlated, and this subject is addressed in Sect. 7.

In terms of nearest neighbors, due to the uncertainty of object locations, the set of actual k nearest neighbors is unknowable. In other words, at any point in time, there may be more than k objects that are possibly the k nearest neighbors. We define a CPkNN (continuous possible kNN) query which is to continuously find all the objects that are possibly the k nearest neighbors. The query returns a sequence of answer-pairs, namely pairs of the form (I, S) such that I is a time interval and S is a set of objects. S may contain more than k objects, but each member x of S may be a k nearest neighbor. In other words, for each object x of S, there is an assignment of locations for all objects (including x) within their uncertainty regions such that when these objects are at these assigned locations, then x is a k nearest neighbor. And conversely, if x is not in S, then regardless where the objects are within their uncertainty regions, x cannot be a k nearest neighbor.

The answer-pair (I, S) provided by our algorithm can be further refined to provide quantitative probability values

under the assumptions that the probability that an object is at a point in its uncertainty region is distributed according to the Gaussian distribution and that these distributions are independent for different objects. In this refinement step, we can output a set of pairs of the form (o, p) where o is an object in the set S and p is the probability that o is a member of the k nearest neighbors. Obviously, output can be limited to objects whose probability exceeds a given threshold. This refinement step can be carried out using the approach given in Cheng et al. [13,28]. This approach is as follows. Assume that S has m objects, where $m \geq k$. For an object o, the probability that o is among the k nearest neighbors is computed as follows. At any instance of time, the locations of the m objects are specified by a $2m$-dimensional vector giving the x, y-coordinates of each of the m objects. The set of all such vectors is a $2m$-dimensional space. In other words, each point in this space is a vector of locations, one for each of the m objects. The method identifies a region R in the $2m$-dimensional space so that object o is among the k nearest neighbors exactly when the locations of the m objects form a point in the region R. Then, p is the probability that the object-locations are in the region R.

As suggested in the above discussion, the uncertain query semantic is applicable in a wide range of scenarios. However, there are also scenarios in which the certain query semantic is applicable. For example, consider CkNN queries that pertain to future, or planned, locations of fleet vehicles (e.g., a FedEx fleet). In this case, the motion plans of each vehicle are known a priori, whereas the uncertainty may not be known in advance. Thus, the future locations of a vehicle, which are derived from its motion plan, are often treated as precise locations in queries.

Processing model: We consider two different processing models, namely off-line processing and on-line processing. Off-line processing assumes that the complete object-trajectories are given a priori, and it computes all the answers. A database update is an insertion of a new object, a deletion of an existing object, or a velocity-vector change of an existing object. For each update the off-line processing re-computes all the answer-pairs. In contrast, on-line processing assumes that trajectories are given dynamically and incrementally. That is, the processing algorithm only knows the trajectory of each object up to the present point in time. Each update extends a trajectory, starts a new trajectory, or ends a trajectory. The processing algorithm returns the current answer whenever it changes. When a trajectory-update is received, the on-line processing only re-computes the current answer. Online processing can be used to monitor real-time situations, such as nearest neighbors in road network and battlefields. Offline processing can be used to analyze historical locations (e.g., clustering historical data streams [31]) or future locations (e.g., locations derived from motion plans).

In the above, we discussed various models in terms motion, certainty, and processing, respectively. In real applications, these models are combined for handling a particular situation. For example, when analyzing historical locations, a suitable combination may be offline processing, piecewise linear motion, and uncertain locations. When monitoring real-time locations, a suitable combination may be online processing, linear motion (until an update is received), and uncertain locations. In applications analyzing motion plans, if the uncertainty is unknown or unimportant, a suitable combination is certain locations and offline processing of piecewise linear motion.

In 1.1.1 we discuss, for the various dimensions of the problem, the value of $\beta_k(n)$ which represents the maximum number of pairs in an answer to the CkNN problem with n moving objects. In 1.1.2, we discuss the algorithms for the various dimensions of the problem. The analysis of these algorithms uses $\beta_k(n)$.

1.1.1 Number of answer-pairs

In the first part of the paper, we study the value of $\beta_k(n)$ which represents the maximum number of pairs in an answer to the CkNN problem with n moving objects. This enables complexity analysis for CkNN query processing in the second part of the paper. Throughout the paper, we assume that k is a constant. We study both the lower bound and the upper bound of $\beta_k(n)$.

We start with the linear motion model. For this model, the tightest upper bound that is known up to date is $8k(n-\mathrm{k})+1$ (see Theorem 3.1 in [20]). In this paper, we give an exact upper bound of $k(2n-k-1)+1$. Since $k(2n-k-1) < 8k(n-k)$ when $k \leq 6n/7$, which holds for most real applications, our bound is tighter.

Now the question is whether the $\mathrm{O}(n)$ upper bound is attainable. In other words, what is the lower bound of $\beta_k(n)$. In this paper, we show that $\mathrm{O}(n)$ is also the lower bound of $\beta_k(n)$. We show this by constructing a feasible configuration in terms of the motion of objects such that $\beta_k(n)$ is equal to $2(n-k)+1$.

Then, we bound $\beta_k(n)$ for the piecewise linear model. For this model, the tightest known upper bound for $\beta_k(n)$ is $\mathrm{O}(nm2^{\alpha(nm)})$ (see Theorem 2.4 in [8]) where α is the functional inverse of Ackermann's function (see [8] for the definition of Ackermann's function). In this paper, we prove a tighter bound of $\mathrm{O}(nm\alpha(n))$. We further show that the lower bound of $\beta_k(n)$ for the piecewise linear model is $2m(n-k)+1 = O(nm)$. Given the fact that $\alpha(n)$ is at most 4 for any practical value of n, the bound of $\mathrm{O}(nm\alpha(n))$ is nearly tight as it only adds an almost-constant factor to the lower bound.

Then, we examine CPkNN, i.e., CkNN under the condition that there is uncertainty associated with the locations of

moving objects. Our analysis shows that the bound on $\beta_k(n)$ in the uncertain case is $O(n)$ times the corresponding bound for the certain case. For example, the bound for the linear model is $\Theta(n^2)$ in comparison with $\Theta(n)$ in the certain case; the bounds for the piecewise linear model are $O(n^2 m\alpha(n))$ and $\Omega(n^2 m)$ in comparison with $O(nm\alpha(n))$ and $\Omega(nm)$ in the certain case.

1.1.2 CkNN algorithms

In the second part of the paper, we use the results of the first part to study the processing of C*k*NN queries in the certain and uncertain cases, respectively. For off-line processing in the certain case, there is an existing simple divide-and-conquer algorithm which gives the solution for the linear model in $O(n\log n)$ time (see Theorem 2.6 in [8]). For on-line processing in the certain case, we develop a kinetic data structure, called *object heap*. This data structure allows updates like insertion of a new object, deletion of an existing object, and velocity-vector change of an existing object. We analyze the complexity of object heap under two different conditions, depending on whether there are updates or not. In either case, we assume that initially each object moves linearly until updated. When there are no updates, the *cumulative complexity* of object heap, i.e., the total cost for returning all answer-pairs (Recall that on-line processing returns one answer-pair each time), is $O(n\log^2 n)$. This is the same as that of *kinetic tournament* [5] and *kinetic heap* [5,6], two classical kinetic data structures for monitoring the nearest neighbor (i.e., k=1)[1]. However, object heap is better at handling updates. In object heap, insertion, deletion, and velocity-vector change can be carried out in $O(\log n)$ time each. In kinetic tournament and kinetic heap, these operations require $O(\log^2 n)$ time each. We prove that if each object can change its velocity-vector at most m times, then the cumulative complexity of our on-line algorithm is $O(nm\alpha(n)\log^2 n)$ which is higher than the lower bound by a factor of $\alpha(n)\log^2 n$. The theoretical results of this paper are summarized in Table 1.

Then, we compare experimentally different algorithms using a database of 1 million objects derived from real-world GPS traces. The results are shown in Fig. 14 through Fig. 25 in Sect. 6. These results show that even though object heap is better than kinetic tournament by a factor of $\log(n)$ in the worst case, it is equally efficient as the latter one for the data set we tested, in the average case. The results also show that on-line processing is more efficient than off-line processing for practical usage. The experimental results also show that the number of answer-pairs grows almost linearly with number of objects for the piecewise linear certain case. They also show how the number of answer-pairs grows in the uncertain case.

In summary, the main contributions of this paper are the following:

1. We bound the number of answer-pairs in a C*k*NN query for various scenarios in terms of the motion model and whether there is uncertainty associated with locations. Some of the bounds that we provide have not been studied before. The others are tighter than existing bounds.
2. We introduce an algorithm called object heap for on-line processing of C*k*NN queries. The worst-case complexity of object heap is lower than that of existing on-line algorithms by a factor of $\log(n)$.
3. We develop an uncertain version of object heap.
4. We evaluate the scalability of the proposed algorithms by experiments using real-world data.

1.2 Comparison with other relevant work

Relationship with other indexing work We discuss each result listed in Table 1 in terms of its applicability in disk storage databases and its combination with indexing structures. First, we consider the certain case. All the results for the number of answer-pairs are applicable to both disk storage databases and main memory databases, regardless of whether indexing structures are used or not. For off-line processing, there are two cases depending on whether the database is disk based or main memory based. In the case of disk storage databases, query processing typically requires a combination of filtering and refinement stages [12,24–26]. In the filtering stage, indexing structures are used to prune the objects that are not candidates to be in the answer set. In the refinement stage, the candidate objects are examined in main memory, and our off-line processing algorithm applies to the refinement stage. In the case of main memory databases, our algorithm is applicable verbatim. Now consider on-line processing. Our on-line processing algorithm (i.e., object heap) is not directly applicable to disk storage databases as is. In the case of main memory databases, the object heap provides indexing with worst-case guarantees, whereas spatial indexing methods such as R-tree do not.

The above discussion applies verbatim to the uncertain case as well.

CkNN in the certain case. The processing of C*k*NN queries has been extensively studied in the database community (see e.g., [3,4,16–18]). However, existing studies lack complex-

[1] Better complexity is known for kinetic heap when the distance functions are pseudo-lines (i.e., any pair of distance functions intersect each other at most once) (see [6]). In our problem, the square-distance functions are parabolas. Observe that each parabola can be cut into two pseudo-lines. However, this will make the square-distance functions partially defined, whereas the analysis in [6] assumes totally defined functions.

Table 1 Summary of the results

Problem case				Our result	Previous result
Certain	Number of answer-pairs	Linear	Lower	$2(n-k)+1$	Not studied
			Upper	$k(2n-k-1)+1$	$8k(n-k)+1$
		Piecewise linear	Lower	$2m(n-k)+1$	Not studied
			Upper	$O(nm\alpha(n))$	$O(nm2^{\alpha}(nm))$*
	Off-line processing, piecewise linear			$O(nm\alpha(n)\log n)$	$O(nm\log(nm))$*
	On-line processing	Single update		$O(\log n)$	$O(\log^2 n)$
		Cumulative complexity with m velocity-vector-changes per object		$O(nm\alpha(n)\log^2 n)$	Not studied
Uncertain case	Number of answer-pairs	Linear		$\Theta(n^2)$	Not studied**
		Piecewise linear		$O(n^2 m\alpha(n))$, $\Omega(n^2 m)$	
	Off-line processing, piecewise linear			$O(n^2 m\alpha(n)\log n)$	
	On-line processing, cumulative complexity with m velocity-vector-changes per object			$O(n^2 m\alpha(n)\log n)$	

* Results obtained by a straightforward application of existing techniques

** For off-line processing in the piecewise linear model in the uncertain case, [14] gives an upper bound of $O(n^3 m)$ for a more complex query semantics

ity analysis due to a limited understanding of the maximum number of answer-pairs. These studies either do not provide algorithm complexity analysis, or treat the number of answer-pairs as an input size that is independent of n. In this paper, we provide a thorough analysis of the maximum number of answer-pairs, for both the certain case and the uncertain case. Our results enable complexity analysis of CkNN algorithms, as demonstrated in this paper.

Li et al. [3] introduce a CkNN algorithm called Beach-line. The algorithm monitors the k-th NN (so called beach-line) since the necessary condition of changes in the kNN set is the change of the kth NN for CkNN. The paper does not provide complexity analysis. Iwerks et al. [4] introduce a C1NN algorithm, the idea of which is similar to the Beach-line algorithm. In that work, the authors present a complexity bound of their algorithm in terms of the number of answer-pairs as an additional parameter. Using our results on the number of answer-pairs (Theorem 2.1), it can be shown that the complexity of the Beach-line algorithm is $O(n^2)$ in the linear model.

Mokhtar et al. [15] proposes a general method of processing moving object queries including CkNN queries. They assume a piecewise linear motion model. They have a general result that a moving object query such as a CkNN query can be processed in polynomial time in terms of the database size. In this paper, we provide a more specific result that a CkNN query can be processed in $O(nm\alpha(n)\log n)$ time in the piecewise linear model.

CkNN queries are studied in [16] in the context where only the query object (i.e., the object that the query pertains to) moves. Since other objects (i.e., data objects) are stationary, they are stored in an R*-tree. Their approach is to use range search to find k closest objects and re-calculate the range at each update on the query object using the moving distance since the last update. This gives a correct query result only at the time of search following the update, and the result may soon become incorrect due to the movement of the query object. In our case, both the query object and the data objects can be moving, and the algorithms analyzed in our paper guarantee the correctness of answers.

Kolahdouzan et al. [17] study CkNN queries in spatial network databases, where the distance between two objects is a function of the network connectivity (e.g., shortest path between two objects). In the present paper, we consider Euclidean distances. Furthermore, in their case, the data objects are static and thus they are able to utilize the Voronoi diagrams to partition the network and facilitate query processing. In our case, the data objects can move. Intuitively constructing and maintaining Voronoi diagrams in this case is inefficient.

Xiong et al. [18] propose an on-line algorithm for CkNN queries where both the query object and the data objects can move. In their model, the actually positions but not velocities of the objects are periodically updated. From the actual positions, the kNN set is computed using an incremental algorithm. On the other hand, we assume that initial locations and velocities are transmitted only once in the linear model, using which the kNN set is computed for all the future time points. Of course we also allow updates to velocities in the piecewise linear model.

In our previous work [19], we studied CkNN in the certain case. We analyzed the number of answer-pairs and developed the object heap data structure for on-line processing. The present paper extends [19] from the following two aspects. First, we study CkNN in the uncertain case by analyzing the number of answer-pairs and developing an algorithm for

on-line processing in this case. Second, we conduct experiments to compare the query processing time of certain case and that of the uncertain case.

Ck NN in the uncertain case. Several studies have been done concerning C*k*NN with location uncertainty. Trajcevski et al [10,14] deal with $k = 1$ only and they consider a different query semantics. In their case, a C1NN query asks continuously for a ranking of objects in the order of their probabilities to be a NN object. They give an upper bound of $O(n^2)$ on the number of answer-pairs in their query semantics for linear motion. We show that this bound is the same in the simpler semantics studied in our paper and it is tight. For piecewise linear motion, they give an upper bound of $O(n^3m)$, whereas in our case it is $O(n^2m\alpha(n))$. Furthermore, [10,14] do not deal with $k > 1$ and on-line processing, whereas we do so in the present paper.

Huang et al. [7,11,12] assume a different uncertainty model, in which the velocity-vector of each moving object follows a probability distribution that is known to query processing. Our uncertainty model does not require the knowledge of the velocity-vector distribution. Furthermore, they do not provide any complexity analysis and they do not deal with on-line processing and piecewise linear model.

Xie et al. [36] study nearest-neighbor queries in which the data objects are represented by static uncertainty regions. In their work, a query is answered by partitioning the geospace using bisectors and computing the intersections between the route of the query object and the partitions. Our problem is more difficult in the sense that the uncertainty regions are also moving. On the other hand, their work supports arbitrary shapes of uncertainty regions, whereas we only consider circular ones.

Probabilistic queries. Probabilistic queries are queries that evaluate data uncertainty and provide probabilistic guarantees [13,27–31]. The answer to a probabilistic *k*NN query lists not only the objects that are possibly the nearest neighbors but also the probability for each of them to be the nearest neighbor. Probabilistic queries are more informative than "qualitative" queries studied in this paper. The probabilistic version of a CP*k*NN query is: "Continuously find all the objects that are possibly the *k* nearest neighbors along with probabilities". However, due to the difficulty of query processing, existing studies are limited to instantaneous queries. Cheng et al. [13,28,30] propose efficient computation and indexing algorithms for evaluating probabilistic *k*NN queries. However, their algorithms only deal with instantaneous rather than continuous queries. In other words, the answer returned by their algorithms only pertains to the locations of the moving objects at the time when the answer is evaluated. The answer does not predict for how long the probability values will be valid. Chen et al. [31] propose algorithms for processing "continuous" probabilistic NN queries. These queries are said to be continuous in the sense that they reside in the database for an extensive amount of time. For these queries, the answer is recomputed whenever there is a location update to the database. The paper develops an incremental evaluation technique so that each update may reuse the query results computed for the previous update. But again, each answer only pertains to the locations at the time when the answer is evaluated.

The rest of this paper is structured as follows: Sect. 2 analyzes the bound on the number of answer-pairs for the certain case. Section 3 analyzes the bound for the uncertain case. Section 4 discusses query processing in the certain case. Section 5 discusses query processing in the uncertain case. Section 6 concludes the paper and discusses some open issues that are worth further study.

2 Continuous kNN with certain location

In this section, we analyze the maximum number of answer-pairs for a C*k*NN query in the certain case. We do this analysis for two reasons. First, the results obtained from this analysis will be used in the analysis for the uncertain case. Second, the results for the uncertain case will become interesting when compared with the results for the certain case.

The rest of this section is organized as follows. In Sect. 2.1, we discuss the number of answer-pairs in the case where objects move linearly. In Sect. 2.2, we discuss the number of answer-pairs in the case where objects move piecewise linearly.

2.1 Number of answer-pairs with linear motion

We consider query processing at a *query object* O_q that is moving. O_q has an *objects database* that stores the motion information of each other *data object* $O_1, O_2, \ldots, O_n$ that is also moving. The *CkNN query* requests for every point in time the k nearest neighbors of O_q among all the data objects. All the objects move in a linear manner in a multi-dimensional space called the *motion space*. That is, each object moves along a straight line with a constant speed. The motion space can have an arbitrarily high number of dimensions. In other words, our results apply to an arbitrary number of dimensions. All the objects start at negative infinity and continue to positive infinity. Then, the square of the distance d_i between O_q and a data object O_i is a quadratic function of time t: $d_i^2(t) = at^2 + bt + c$, where a, b, and c are parameters dependent on the velocities and initial locations of O_q and O_i. The coefficient a is non-negative and thus the $d_i^2(t)$ function is convex. The $d_i^2(t)$ function is a parabola in the Time-Square_Distance space

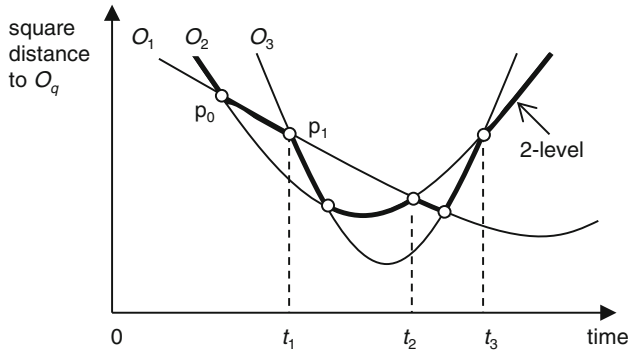

Fig. 3 Example of answer-pairs in the Time-Square_Distance space

[3] as illustrated in Fig. 3. For convenience of presentation, in the rest of this paper we use object O_i and its square-distance function $d_i^2(t)$ interchangeably when there is no confusion.

The CkNN query is issued at time 0. At that time O_q has a set of k nearest neighbors, but as time progresses, the kNN set may change. A time interval during which the kNN set remains unchanged is referred to as an *answer interval*. An *answer-pair* is a pair consisting of an answer interval and its associated kNN set. Answer-pairs are required to be non-redundant in that, for any pairs $(I,\ S)$ and $(I',\ S')$, the sets S and S' are distinct if I and I' are contiguous. As an example, in Fig. 3, if $k = 2$, then the answer-pairs are :

$\langle[0, t_1), \{O_1, O_2\}\rangle$,
$\langle[t_1, t_2), \{O_2, O_3\}\rangle$,
$\langle[t_2, t_3), \{O_1, O_3\}\rangle$,
$\langle[t_3, \infty), \{O_1, O_2\}\rangle$

Denote by $I_k(G)$ the number of answer-pairs for an instance G of the objects database for the CkNN query. Define

$\beta_k(n) = \max\{I_k(G) | G$ is an instance of the objects database with n objects$\}$

In words, $\beta_k(n)$ is the maximum number of answer-pairs for the CkNN query among all the objects database instances with n objects. First, we discuss the upper bound of $\beta_k(n)$. The following definition is used in the discussion.

Definition 2.1 Let p be an intersection of two objects A and B. We use *p.time* to denote the time coordinate of the intersection. p is associated with another two attributes, namely its downward and upward. A is the *downward* of p, and A *downward-crosses* B at p, if A is farther than B immediately before *p.time* (and thus closer than B immediately after *p.time*). In this case, B is the *upward* of p, and B *upward-crosses* A at p.

Clearly, for any two objects A and B, A downward-crosses B at most once and A upward-crosses B at most once.

Now we examine the upper bound of $\beta_k(n)$. Observe that for any objects database instance, the number of answer-pairs is upper bounded by the number of times the k-th nearest-neighbor changes. This is because each change of the answer set is always caused by a change of the k-th nearest neighbor, whereas a change of the k-th nearest neighbor does not necessarily cause a change of the answer set. Specifically, the answer set changes only when there is a switch of order between the k-th nearest neighbor and the $(k + 1)$st nearest neighbor. In the Time-Square_Distance space, such a switch occurs when a k-th lowest curve upward-crosses a curve. For example, in Fig. 3, at point p_1, the 2nd lowest curve O_1 is crossed by O_3 from above, and thus, the answer set is changed at p_1. Specifically, O_1 is removed from the answer set, and O_3 is added to the answer set. On the other hand, the answer set does not change when a k-th lowest curve downward-crosses a curve. For example, in Fig. 3, at point p_0, the 2nd lowest curve O_2 is crossed by O_1 from below, and thus, the answer set is not changed at p_0.

According to the above observation, the number of answer-pairs is upper bounded by the number of pieces in the envelope formed by the k-th lowest curves. The latter number has been studied in the context of arrangements of curves and is formally defined as follows (see [8]). Given a set Γ of curves where each curve is a continuous and univariate function, the arrangement $A(\Gamma)$ of Γ is the planar subdivision induced by the curves in Γ. That is, $A(\Gamma)$ is a planar map the *vertices* of which are the pair-wise intersection points of the curves in Γ and the *edges* of which are maximal connected portions of the curves that do not contain a vertex. As an example, Fig. 3 shows the arrangement of three objects O_1, O_2, and O_3, where white circles are vertices. The *level* of a point p in $A(\Gamma)$ is the number of curves in Γ not above p, and the level of an edge $e \in A(\Gamma)$ is the common level of all the points lying in the relative interior of e[2]. The *k-level* of $A(\Gamma)$ is the union of all edges in $A(\Gamma)$ whose level is k-1. The *length* of the k-level is the number of edges in the k-level. In Fig. 3, the thick curve shows the 2-level in the arrangement of O_1, O_2, and O_3; its length is 7.

It is well known that the length of the k-level in an arrangement of n parabolas, each with an axis of symmetry that is parallel to the y-axis, is $O(n)$ for any constant k (see e.g. [9]). However, most of the existing literature only gives order-statistics results. Theorem 3.1 in [20] implies an exact bound of $8k(n-k)+1$. In the following theorem, we give a tighter exact bound.

[2] In the computational geometry literature, the level of a point p is defined to be the number of curves lying strictly below p. In this paper, we define the level to be the number of curves not above p, so that the k-level corresponds to the k-th nearest neighbor. But this definition is purely terminological without affecting any results.

Theorem 2.1 *Assume that the query object and data objects all move linearly. Then for any constant k* $\beta_k(n) \leq k(2n - k - 1) + 1$.

For the proof of Theorem 2.1, we give the following definition. Given the arrangement of data objects in the Time-Square_Distance space, an intersection point in the arrangement is referred to as a $\leq$*k-level intersection* (respectively $>$*k-level intersection*) if it is an endpoint of an edge the level of which is smaller than or equal to k (respectively greater than k).

Proof of Theorem 2.1 We prove by showing that the number of $\leq k$-level intersections is at most $k(2n - k - 1)$. Without loss of generality, we assume that at the time when the query is processed, O_1 is the first nearest neighbor, O_2 is the second nearest neighbor, …, and O_n is the n-th nearest (and the farthest) neighbor (see Fig. 4a). We construct the arrangement in the same order. That is, we add O_1 first, and then add O_2, and so on. We show that for any integer $k < j \leq n$, the addition of O_j can introduce at most $2k \leq k$-level intersections to the final arrangement of the n objects. To do this, we divide the intersections introduced by O_j to the arrangement of the first j objects into two groups, namely the $\leq k$-level intersections and $> k$-level intersections. We call these two groups *lower intersections* and *higher intersections* respectively. Observe that, due to the later introduction of O_{j+1}, O_{j+2}, and so on, some of the lower intersections may become $> k$-level intersections in the final arrangement. However, none of the higher intersections may become $\leq k$-level intersections in the final arrangement. Thus, the number of $\leq k$-level intersections introduced by O_j to the final arrangement is upper bounded by the number of lower intersections. In the following, we show that the number of lower intersections is at most $2k$.

If O_i never reaches the k-th position, then the statement clearly holds. If O_j does reach the k-th position, then consider the first time it reaches the $(k + 1)$st position. Denote that time by T. Observe that O_j has to experience a series of downward-crosses to reach the $(k + 1)$st position. We refer to the objects that are closer to O_q than O_j at time T as the *lower objects* and the objects that are farther away from O_q at time T as the *higher objects*. Clearly, there are k lower objects. In Fig. 4b, the objects below O_j are the lower objects. The objects above O_j, including O_{j+1}, O_{j+2}, …, O_n, are the higher objects. Notice that the set of lower objects and that of higher objects pertain to time T.

Assume that object O_j participates in a certain number of lower intersections with the higher objects after time T. Denote this number by d. Obviously, O_j can make at most $2k$ lower intersections with the lower objects. Observe that for each lower intersection that O_j makes with an higher object after time T, which higher object will move below O_j and it can never move above O_j again, because the downward-cross has already been consumed before time T.

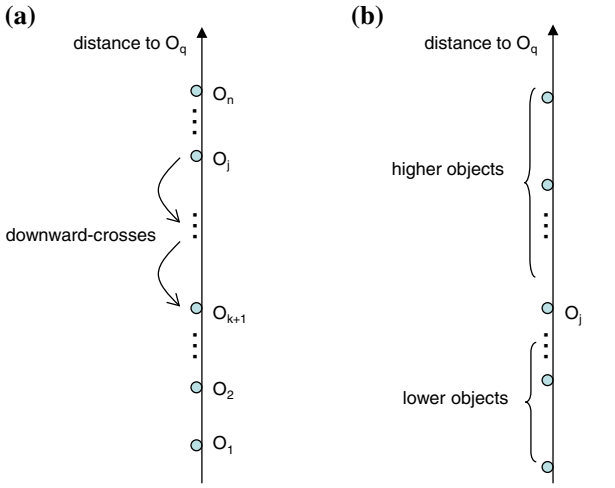

Fig. 4 The auxiliary figure for the proof of Theorem 2.1. **a** Initial configuration. O_j has to experience a series of downward-crosses to reach the $(k + 1)$st position. **b** At the time when O_j reaches the $(k + 1)$st position for the first time, the objects below it are lower objects, and the objects above it (excluding $O_{j+1}, O_{j+2}, \ldots, O_n$) are higher objects

Now observe that if more than $k - d$ of the lower objects make 2 lower intersections with O_j, it means that an higher object upward-crosses O_j after time T; thus this is impossible. Therefore at most $k-d$ of the lower objects make 2 lower intersections with O_j and each one of the rest makes at most one lower intersection with O_j. Thus, if the number of lower intersections of higher objects with O_j is d, then the maximum number of lower intersections of lower objects with O_j is $2k - d$. Thus, the maximum number of lower intersections is $2k$.

So far we have studied the case of $k < j \leq n$. Using similar argument, we can show that in the case of $j \leq k$ the number of $\leq k$-level intersections introduced by O_j to the final arrangement is $2(j - 1)$.

In summary, the total number of $\leq k$-level intersections in the final arrangement is as follows:

$$2k(n - k) + \sum_{j=1}^{k} 2(j - 1) = k(2n - k - 1)$$

The length of the k-level is upper bounded by the number of $\leq k$-level intersections plus one. Thus, the number of answer-pairs is at most $k(2n - k - 1) + 1$. □

For some k's, the exact number given by Theorem 2.1 is tight. For example, when $k = 1$, the number of answer-pairs is equal to the length of the 1-level which is tightly bounded by $2n - 1$ according to Theorems 2.1 and 3.1 in [8]. Theorem 2.1 gives the same bound in this case. The order

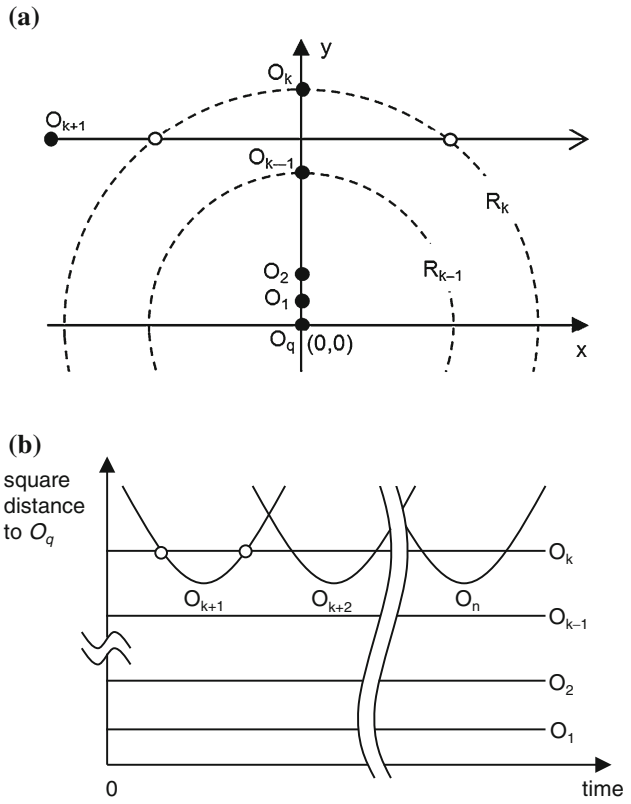

Fig. 5 The auxiliary figure for the proof of Proposition 2.2. **a** Configuration in the motion space. **b** Arrangement in the Time-Square_Distance space

given by Theorem 2.1 (i.e., O(n)) is tight for all values of k, as shown by the following proposition.

Proposition 2.2 *Assume that the query object and data objects all move linearly. Then for any constant k* $\beta_k(n) \geq 2(n-k)+1$.

Proof We construct a feasible case in which the number of answer-pairs is linear in n. Let objects O_q, O_1, O_2, ..., and O_k be static in a 2D plane of the motion space such that O_i is the i-th nearest neighbor of O_q, as shown in Fig. 5a. Denote by R_i the circle the center of which is the location of O_q and the radius of which is the distance between O_i and O_q. Let object O_{k+1} move in the same 2D plane such that its route intersects R_k twice but does not intersect R_{k-1}. Let O_{k+2} have the same route as O_{k+1} and move behind O_{k+1} such that it enters R_k after O_{k+1} leaves R_k. Construct the same for O_{k+3} and so on. Figure 5b shows the arrangement of the objects in the Time-Square_Distance space. It is easy to see that the number of answer-pairs is $2(n-k)+1$. □

Corollary 2.3 *Assume that the query object and the data objects all move linearly. Then for any constant k* $\beta_k(n) = \Theta(n)$.

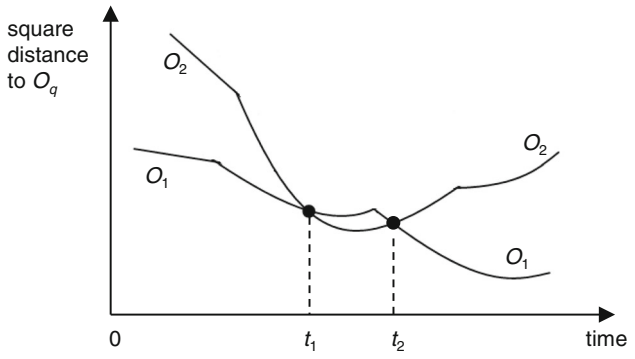

Fig. 6 In the piecewise linear model each object is represented by a connected sequence of parabola pieces

2.2 Number of answer-pairs with piecewise linear motion

Now we assume that objects move piecewise linearly in the motion space. That is, an object moves along a straight line with a constant speed from point a to point b. At point b the object changes its velocity-vector and moves to point c; there it changes the velocity-vector again and moves to point d, etc. Observe that in the Time-Square_Distance space in this model each object is represented by a connected sequence of parabola-segments (see Fig. 6). It is easy to see that if the motion of O_q has m linear pieces and that of each object O_i also has m linear pieces, then O_i has at most $2m$-1 parabola-segments in the Time-Square_Distance space.

According to Theorem 2.5 in [8], the length of the k-level in an arrangement of N parabola-segments is $O(N2^{\alpha(N)})$ where α is the functional inverse of Ackermann's function[3]. Based on this result, $\beta_k(n) = O(nm2^{\alpha(nm)})$ because there are at most $n(2m-1)$ parabola-segments totally for all objects in the Time-Square_Distance space. However, this derivation treats the parabola-segments of each individual object as independent segments. It does not utilize the fact that they are one-by-one connected. With the connectivity property taken into account, we obtain a tighter bound in the following theorem.

Theorem 2.4 *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces. Then* $\beta_k(n) = O(nm\alpha(n))$.

Due to the extremely fast growth of Ackermann's function, its inverse $\alpha(n)$ grows extremely slowly and is at most 4 for any practical value of n.

Proof idea The proof is inspired by Corollary 3.4 in [8]. The Corollary implies that the length of the k-level in an arrangement of n piecewise linear functions, where each function has m linear pieces, is $O(nm\alpha(n))$. It can be shown that Corollary 3.4 in [8] holds for piecewise pseudo-linear functions as well.

[3] See [7] for the definition of Ackermann's function.

Here, pseudo-linear functions are defined to be unbounded curves such that any two of them intersect at most once. We then cut each parabola into two pseudo-linear pieces using its axis of symmetry. Thus, the $O(nm\alpha(n))$ bound follows. □

The following proposition gives a lower bound of $\beta_k(n)$ in the piecewise linear model.

Proposition 2.5 *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces. Then for any constant k* $\beta_k(n) \geq 2m(n-k)+1$.

Proposition 2.5 tells us that the bound of $O(nm\alpha(n))$ is very tight because it only adds an almost-constant factor to the lower bound.

Proof idea Similar to the linear case, we construct a feasible case in which the number of answer-pairs is linear in nm. For details, see Appendix A which is provided in Electronic supplementary material. □

Corollary 2.6 *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces.* $\beta_k(n) = O(nm\alpha(n))$ *and* $\beta_k(n) = \Omega(nm)$.

3 Continuous kNN with location uncertainty

Our uncertainty model is a circular region, and the object may be anywhere in this region. This captures the uncertainty caused by positioning errors. It is also a result of the widely used "dead-reckoning" location update policy [23] which reduces the communication/energy cost for a moving object to report its location changes. In this policy, a moving object sends an update to the database when the distance between its current location and the location stored in the database exceeds a certain threshold. Thus, at any point in time, the actual location of the moving object can be anywhere in a circular region with the center being the location stored in the database and the radius being the threshold.

The rest of this section is organized as follows. In Sect. 3.1, we formalize the uncertainty model and define the semantics of continuous kNN query in the context of uncertainty. In Sect. 3.2, we discuss the number of answer-pairs when uncertainty regions move linearly. In Sect. 3.3, we discuss the number of answer-pairs when uncertainty regions move piecewise linearly.

3.1 Uncertainty model and continuous possible kNN queries

For any point t in time, each data object O_i has an *uncertainty region* $U_i(t)$, which is a closed region where the location of O_i can possibly be at t. $U_i(t)$ is a circle area with the center denoted by $C_i(t)$ and the radius denoted by $r_i(t)$ (see e.g., [10, 13]). We assume that the radius $r_i(t)$ is a constant, i.e., does not change with time. However, this constant may be different among objects. We write $r_i(t)$ as r_i. The location of O_q has no uncertainty because O_q knows its own location precisely. A *location configuration at time t* is a sequence of points $p_1, p_2, \ldots, p_n$ such that $p_1 \in U_1(t)$, $p_2 \in U_2(t) \ldots, p_n \in U_n(t)$. Intuitively, a location configuration gives a possible location for each object at a particular time.

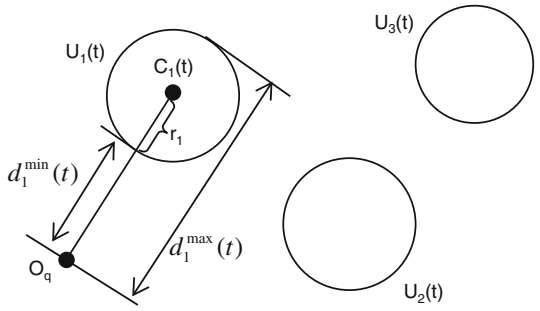

Fig. 7 Uncertainty regions, minimum possible distance, and maximum possible distance

The uncertainty of location values implies that there may be more than k objects that are possibly the k nearest neighbors of O_q at a point in time. This concept is illustrated by Fig. 7 for $k = 1$ (i.e., nearest neighbor). Clearly both O_1 and O_2 are possible nearest neighbors at t. O_3, on the other hand, is not a possible nearest neighbor at t.

Formally, an object O_i is a *possible k nearest neighbor (PkNN) at time t* if there exists a location configuration at t in which O_i is the 1st, or 2nd, or … k-th nearest neighbor. *The PkNN set at time t* is the set of all the objects that are PkNNs at t. For the objects database instance shown in Fig. 7, $\{O_1, O_2\}$ is the P1NN set at time t. The *continuous PkNN (CPkNN) query* requests for every point in time the PkNN set of O_q.

Denote by $d_i^{\max}(t)$ the maximum possible distance between O_i and O_q at time t. $d_i^{\max}(t) = d_i(t) + r_i$ where $d_i(t)$ is the distance between $C_i(t)$ and the location of O_q at t. $(d_i^{\max}(t))^2$ is a parabola in the Time-Square_Distance space as illustrated in Fig. 8 by solid curves. The $(d_i^{\max}(t))^2$ curve is referred to as *the maximum square-distance curve, or simply the max-curve, of* O_i. Similarly, the minimum possible distance between O_i and O_q at time t is $d_i^{\min}(t) = d_i(t) - r_i$. $(d_i^{\min}(t))^2$ is also a parabola in the Time-Square_Distance space as illustrated in Fig. 8 by dashed curves[4]. The $(d_i^{\min}(t))^2$ curve is referred to as the *minimum*

[4] Strictly speaking, the minimum possible distance is zero when $d_i(t) - r_i \leq 0$, which occurs when the location of O_q at t falls into the uncertainty region $U_i(t)$. In this case, the min-curve is a parabola truncated by the horizontal line y=0. However, for simplicity of

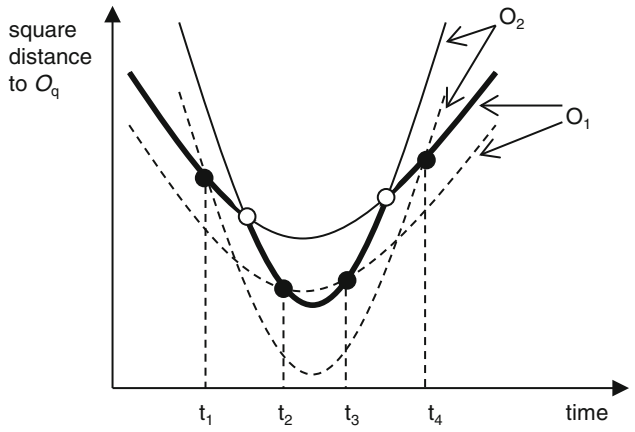

Fig. 8 Max-curves (*solid parabolas*) and min-curves (*dashed parabolas*)

square-distance curve, or simply the *min-curve*, of O_i. Thus, each data object O_i is represented by a pair of parabolas namely $(d_i^{\max}(t))^2$ and $(d_i^{\min}(t))^2$.

Example 1 Consider the CP1NN query against the objects database instance shown in Fig. 8. The answer-pairs are as follows:

$< [0, t_1), \{O_1\} >,$
$< [t_1, t_2), \{O_1, O_2\} >,$
$< [t_2, t_3), \{O_2\} >,$
$< [t_3, t_4), \{O_1, O_2\} >,$
$< [t_4, \infty), \{O_1\} >$

We refer to the k-level in the arrangement of the max-curves as the *max k-level*. The thick solid curve in Fig. 8 shows a max 1-level.

The following proposition provides a method to determine a PkNN set at a given point in time. It is a standard technique used in existing work on PkNN query processing (see [7,11,12]).

Proposition 3.1 *For any k, an object O_i is a PkNN at a time t if and only if the min-curve of O_i is below the max k-level at t.*

The intuition behind Proposition 3.1 is as follows. Suppose that the min-curve of O_i is above the max k-level at time t. Consider the objects that are the max 1-level, max 2-level,..., the max k-level at t. These k objects are definitely closer to O_q than O_i regardless of the location configuration. Thus, in this case, O_i is not a PkNN at t. Now suppose that the min-curve of O_i is below the max k-level at time t. Then, there are at least $n-k$ objects such that their max-curves are above O_i's min-curve. Let it be such that for each of these $n-k$ objects its location takes its max-curve at t, and that the location of O_i takes its min-curve at t. In this case O_i is a kNN at t.

According to Proposition 3.1, the PkNN set changes if and only if a min-curve intersects the max k-level. Specifically, if the min-curve downwards crosses the max k-level at a time t, then O_i should be added to the PkNN set starting from t. On the other hand, if the min-curve upwards crosses the max k-level at a time t, then O_i should be removed from the PkNN set starting from t.

As shown in the Fig. 8 example, a PkNN set may contain more than k objects. If size of the set is much larger than k, then the user may request that the PkNNs be ranked by their probabilities of being a k nearest neighbor for a particular time point. In this case, the technique presented in [10,14] can be used for ranking.

3.2 Number of answer-pairs for CPkNN queries with linear motion

Denote by $\tilde{I}_k(G)$ the number of answer-pairs for an instance G of the objects database for the CPkNN query. Define

$\tilde{\beta}_k(n) = \max\{\tilde{I}_k(G) | G$ is an instance of the objects database with n objects$\}$

In words, $\tilde{\beta}_k(n)$ is the maximum number of answer-pairs for the CPkNN query among all the objects database instances with n objects.

We assume that the query object O_q moves linearly. In addition, for each data object O_i, its expected location $C_i(t)$ also moves linearly. It is obvious that $\beta_k(n)$ is upper bounded by $O(n^2)$ because there are $O(n^2)$ intersections among all the min-curves and all the max-curves. The following theorem indicates that $O(n^2)$ is also the lower bound of $\beta_k(n)$.

Theorem 3.2 *Assume that the query object moves linearly and that each data object's expected location moves linearly. Then for any constant k $\tilde{\beta}_k(n) = \Omega(n^2)$.*

Proof Prove by constructing a feasible case in which the number of answer-pairs is quadratic in n. The construction proceeds as follows. Let object O_q and the expected locations of $O_1, O_2, \ldots$, and O_k be static in the motion space such that O_i has the i-th maximum possible distance to O_q (see Fig. 9a). Denote by $R_i^{\max}$ the circle the center of which is the location of O_q and the radius of which is the maximum possible distance between O_i and O_q. Let object O_{k+1} move such that the trace of its farthest-to-O_q-point intersects $R_k^{\max}$ twice but does not intersect $R_{k-1}^{\max}$, as shown in Fig. 9a. Let O_{k+2} have the same route as O_{k+1} and move behind O_{k+1} such that its farthest-to-O_q-point enters $R_k^{\max}$ after that of

Footnote 4 continued
presentation, we assume that $d_i(t) - r_i$ is greater than zero for all values of t. But all the results in this paper hold for the case where $d_i(t) - r_i$ may be smaller than or equal to zero.

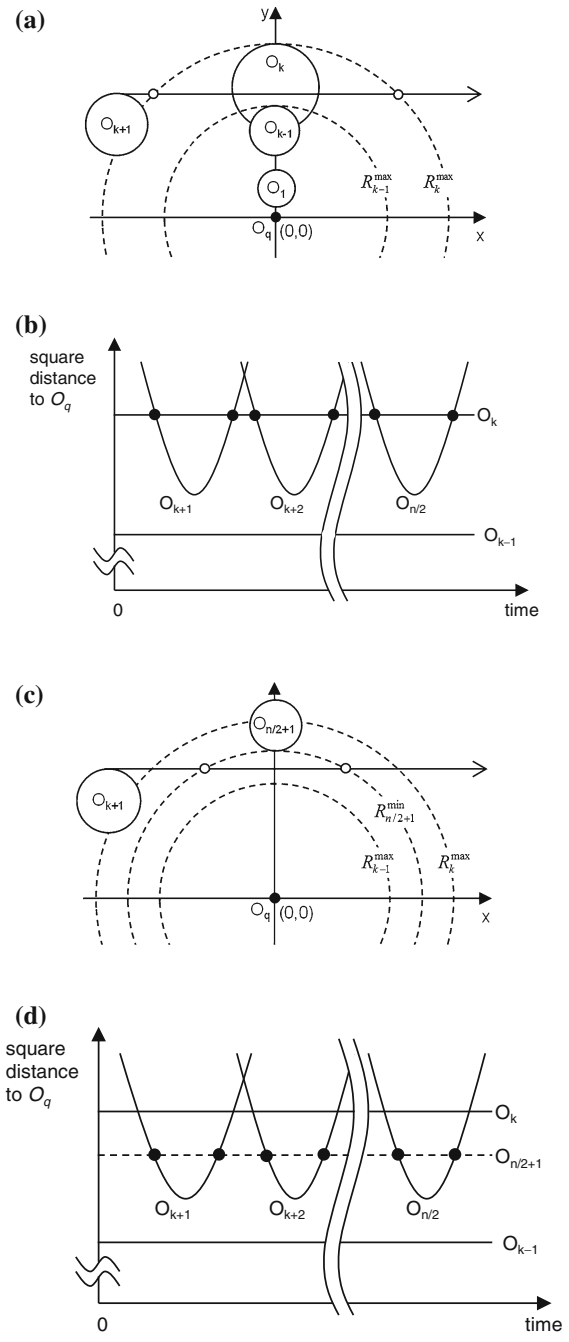

Fig. 9 The auxiliary figure for the proof of Theorem 3.2. **a** Configuration of the first $n/2$ objects in the motion space. **b** Max-curves of the first $n/2$ objects. **c** Configuration of $O_{n/2+1}$ in the motion space. **d** Intersections between the min-curve of $O_{n/2+1}$ and the max k-level

O_{k+1} leaves $R_k^{\max}$. Construct the same for $O_{k+3}, \ldots, O_{n/2}$. Figure 9b shows the arrangement of the max-curves of the first $n/2$ objects.

Denote by $R_i^{\min}$ the circle the center of which is the location of O_q and the radius of which is the minimum possible distance between O_i and O_q. Let $O_{n/2+1}$ be static such that $R_{n/2+1}^{\min}$ twice intersects the trace of O_{k+1}'s farthest-to-O_q-point (see Fig. 9c). Figure 9d shows how the min-curve of $O_{n/2+1}$ intersects the max k-level. As shown in figure, the max k-level contains $\frac{n}{2} - k$ parabola pieces, contributed by O_{k+1}, $O_{k+2}, \ldots, O_{n/2}$ respectively. There are totally $2 \cdot (\frac{n}{2} - k)$ intersections between the min-curve of object $O_{n/2+1}$ and these parabolas, each intersection being one critical intersection. Do the same construction for $O_{n/2+2}, \ldots, O_n$. By the same reasoning for object $O_{n/2+1}$, for each of these objects, there are $2 \cdot (\frac{n}{2} - k)$ critical intersections. Thus, there are totally $2 \cdot \frac{n}{2} \cdot (\frac{n}{2} - k)$ critical intersections. □

Corollary 3.3 *Assume that the query object moves linearly and that each data object's expected location moves linearly. Then for any constant k $\tilde{\beta}_k(n) = \Theta(n^2)$.*

Corollary 3.3 is particularly interesting when compared with the result in the certain case: the number of answer-pairs is linear in n in the certain case (see Theorem 2.1), whereas it is quadratic in n in the uncertain case.

3.3 Number of answer-pairs for CPkNN queries with piecewise linear motion

In this subsection, we derive the upper bound and lower bound of $\tilde{\beta}_k(n)$ for the piecewise linear model. Observe that in the piecewise linear model, each max-curve is a connected sequence of parabola curves, and so is each min-curve. The following theorem gives an upper bound of $\tilde{\beta}_k(n)$.

Theorem 3.4 *Assume that the query object and the expected locations of each data object move piecewise linearly, where each object has at most m pieces. Then for any constant k $\tilde{\beta}_k(n) = O(n^2 \cdot m \cdot \alpha(n))$.*

The proof of Theorem 3.4 is based on the following lemma which bounds the number of intersections between two connected sequences of parabola pieces. A connected sequence of parabola pieces is a sequence of parabola pieces such that the ending point of one piece is the starting point of its successive piece. An obvious bound for the number of intersections between two sequences parabola pieces, where one sequence has m_A pieces and another sequence has m_B pieces, is $O(m_A \cdot m_B)$. However, the following lemma gives a much tighter bound which is $O(m_A + m_B)$.

Lemma 3.5 *Let A be a connected sequence of m_A x-monotone parabola pieces and B be a connected sequences of m_B x-monotone parabola pieces, in the Time-Square_Distance space. There are at most $2(m_A + m_B - 1)$ intersections between A and B.*

Lemma 3.5 is proved in Appendix B which is included in Electronic supplementary material. Now we prove Theorem 3.4.

Proof of Theorem 3.4 According to Theorem 2.4, the length of the max k-level is $O(n \cdot m \cdot \alpha(n))$. Each min-curve has at most $2m - 1$ pieces. According to Lemma 3.5, the number of intersections between each min-curve and the max k-level is upper bounded by

$$2 \cdot ((2 \cdot m - 1) + O(n \cdot m \cdot \alpha(n)) - 1) = O(n \cdot m \cdot \alpha(n))$$

Thus, the total number of intersections between all the min-curves and the max k-level is $n \cdot O(n \cdot m \cdot \alpha(n)) = O(n^2 \cdot m \cdot \alpha(n))$. □

The following theorem gives a lower bound of $\tilde{\beta}_k(n)$.

Theorem 3.6 *Assume that the query object and the expected locations of each data object move piecewise linearly, where each object has at most m pieces. Then for any constant k* $\tilde{\beta}_k(n) = \Omega(n^2 \cdot m)$.

We prove the theorem by constructing a feasible case in which the number of answer-pairs is in order of $n^2 \cdot m$. For details, see Appendix C which is included in Electronic supplementary material.

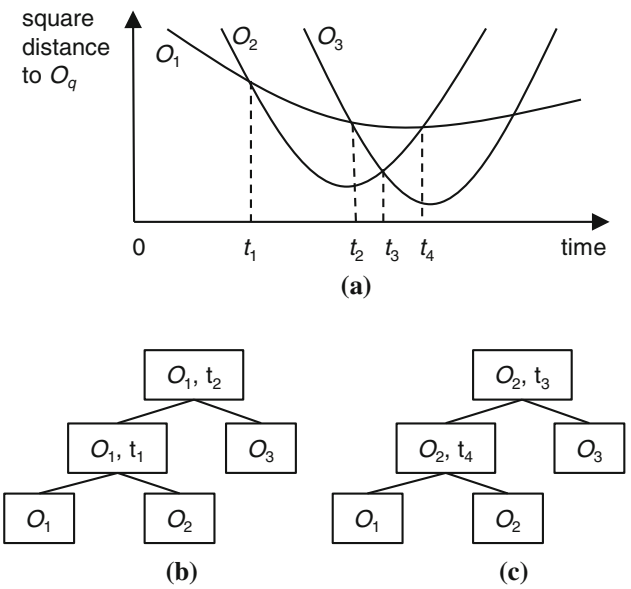

Fig. 10 Kinetic tournament

4 Query processing without uncertainty

In this section, we discuss the processing of CkNN queries without uncertainty. In Sect. 4.1, we discuss off-line processing and in Sect. 4.2, we discuss on-line processing.

4.1 Off-line processing

When $k = 1$, the processing of a CkNN query reduces to the construction of the lower envelope in the arrangement of the square-distance functions of the n data objects (i.e., the $d_i^2(t)$'s). In the linear model, the lower envelope can be constructed using a simple and efficient divide-and-conquer algorithm in O($n$log$n$) time (see Theorem 2.6 in [8]). This algorithm divides the objects database into two subsets S_1 and S_2, each of size at most $\lceil n/2 \rceil$, computes the lower envelopes of S_1 and S_2 recursively, and merges the two envelopes to obtain the lower envelope of the objects database.

The same paradigm can be used to compute the lower envelope in the piecewise linear model. In this case, merging the lower envelopes of S_1 and S_2 still takes time proportional to the sum of $|S_1|$ and $|S_2|$ as described in [8]. Due to Theorem 2.4, $|S_1| + |S_2| = \mathrm{O}(nm\alpha(n))$. Thus, the complexity in the piecewise linear model is $T(n) = 2T(\frac{n}{2}) + O(nm\alpha(n))$, which is O($nm\alpha(n)logn$).

When $k > 1$, the answer-pairs can be computed in the same fashion. Details are omitted due to space limitations.

4.2 On-line processing

In Sect. 4.2.1, we describe an existing on-line processing data structure called *kinetic tournament* and discuss its shortcoming. From 4.2.2 to 4.2.5, we present our on-line processing data structure which overcomes this shortcoming.

4.2.1 Kinetic tournament

Consider the case in which $k = 1$ and all the objects move linearly. The query processing in this case translates to the problem of dynamically maintaining the lower envelope of a set of parabolas in the Time-Square_Distance space. In computation geometry, this problem is known as *kinetic minimum maintenance*. The authors of [5] introduce a solution to this problem, which is called a *kinetic tournament*. The idea is to use a simple divide-and-conquer strategy. The algorithm partitions the data objects into two approximately equal-sized groups (arbitrarily), and recursively maintains the minimum of each group. A final comparison at the top level yields the global minimum. If viewed from the bottom up, this is exactly a tournament for computing the global winner. Each comparison of two data objects is associated with an event that describes when this comparison will be violated in the future. When a violation happens, the new winner is produced and is percolated up the tournament tree, until it is either defeated or declared the global winner.

Figure 10 gives an example of kinetic tournament. Figure 10a shows three data objects O_1, O_2, and O_3 in the Time-Square_Distance space. Figure 10b shows the tournament tree constructed at time 0. Two events are scheduled at this moment: (i) O_1 crosses O_2 at time t_1 which represents that O_1 beats O_2 up to t_1; (ii) O_1 crosses O_3 at time t_2 which represents that O_1 beats O_3 up to t_2. Since $t_1 < t_2$, the O_1-crossing-O_2 event is triggered earlier, at t_1. In response to this event, the winner between O_1 and O_2 is changed, and the

new winner is percolated up (see 3.1(c)). The O_1-crossing-O_3 event is eliminated, and two new events are scheduled to be triggered, at time t_4 and t_3 respectively.

Now examine the complexity of kinetic tournament on processing each event. The processing of an event involves the percolation of the new winner. Because a tournament tree is balanced, the percolation visits at most O(log$n$) nodes. A visit to each of these nodes may result in elimination of an existing event and creation of a new event. If we use a priority queue to store the relevant events (at most O(n)), then the elimination or creation of an event takes O(log$n$) time. Thus, the percolation takes O(log^2n) time.

As analyzed above, in kinetic tournament, O(log$n$) time is spent on accessing the event queue when processing an event. Indeed, kinetic tournament schedules an event for each internal node in the tournament tree, even though some of these events will never be triggered. In the Fig. 10 example, the O_1-crossing-O_3 event is never triggered because it becomes useless after O_2 claims the global winner. In fact we only need to schedule one event which is the earliest time at which the current global winner may be replaced. In this way, we eliminate the cost of accessing the event queue. This is the main idea of our on-line processing data structure which is presented in the following subsection.

4.2.2 The object heap data structure

Our kinetic data structure is called *object heap* and is organized as follows. As before, we assume that we have a query object O_q and a set S of data objects such that $O_q \notin S$. We assume that there is a unique id with each object in S. Let $t \geq 0$ be a time instance. An object heap H over the set S at time t is a full binary tree such that each node x in it stores two items $x.object$ and $x.time$ satisfying the following conditions. First, for any internal node x, let $ObjectSet(x)$ denote the set of objects stored at the leaves of the sub-tree rooted at x. For a leaf node x, $x.time = \infty$. For an internal node x, $x.time \geq t$ and $x.object$ is the closest object to O_q during the time interval $[t, x.time]$ among all the objects in $ObjectSet(x)$. For any internal node x, we let $x.left$, $x.right$ represent, respectively, the left and right child of x. Also, $x.parent$ represents the parent of x. Figure 11 shows the object heap at time 0 for the arrangement of Fig. 10a. From the heap, we see that O_1 is the closest object until t_1.

An object heap is constructed using the procedure *BuildObjectHeap*(S, O_q, $Ctime$) given below. The procedure takes the set of objects S, the query object O_q and time $Ctime$ and builds an object heap at time $Ctime$. $CloserObject(O_1, O_2, t)$, $NextTime(O_1, O_2, t)$ are functions that take two objects O_1, O_2 and time t as parameters and returns values as defined below. $CloserObject(O_1, O_2, t)$ returns the object closest to O_q among the two objects O_1, O_2 at time t. $NextTime(O_1, O_2, t)$ returns the earliest

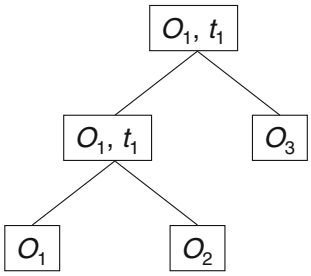

Fig. 11 The object heap at time 0 for the arrangement of Fig. 10a

time $t' > t$ such that $CloserObject(O_1, O_2, t')$ is different from $CloserObject(O_1, O_2, t)$, i.e. when one of the two objects becomes closer than the other; if no such t' exists then it returns ∞. The BuildObjectHeap procedure first initializes the heap so that each object in S is stored at a leaf node. Then, it processes nodes in decreasing order of their levels. For each internal node, it sets its object to be the closest object to O_q at time $Ctime$ among the objects stored at its two children; it sets its time to be the minimum of the time values in its children and the next time when one of them is going to overtake the other to become closer to O_q. In this procedure, min is a function that returns the minimum of three values.

```
BuildObjectHeap(S,Oq,Ctime)
Initialize();
For each node x in decreasing values of the level of x
    If x is an internal node
        O1 := x.left.object;
        O2 := x.right.object;
        x.object := CloserObject(O1, O2, Ctime);
        x.time := min(x.left.time, x.right.time,
        NextTime(O1,O2,Ctime))
```

It is fairly obvious to see that the worst-case time complexity of *BuildObjectHeap*() is $O(n)$ where n is the number of objects in S. Assuming that the start time is zero, the initial object heap is built by invoking the above procedure with the parameter value of $Ctime$ set to zero. It is to be noted that if r is the root node of the object heap then $r.object$ is the closest object until the time $r.time$. Thus, at time $r.time$, the object heap needs to be readjusted. We call such a readjustment as an *implicit update*.

4.2.3 Algorithm for implicit updates

We classify the internal nodes of an object heap as *cross nodes* and *minimal nodes* as follows. An internal node x is called a cross node, if $x.time$ is less than both $x.left.time$ and $x.right.time$. All internal nodes other than cross nodes are called minimal nodes. If x is a cross node and O_1, O_2 are the objects at its children and $x.object = O_1$, then up to the time $x.time$ object O_1 is the closest to O_q among objects in $ObjectSet(x)$ and at time $x.time$, O_2 will be the closest

object among these objects. If x is a minimal node, then there is a child y of x such that $x.time = y.time$.

Roughly speaking, the algorithm for implicit update works as follows. If the root node r is a cross node then it sets $r.object$ to be the object, among its children, which is different from the current value of $r.object$ and sets $r.time$ to be the minimum of the times at its children. If the root node r is a minimal node then it traverses along a path of internal nodes $x_1, \ldots, x_g$ such that $x_1 = r$, x_g is a cross node and all nodes $x_1, \ldots, x_{g-1}$ are minimal nodes and the *time* value on all these nodes is $r.time$. After reaching x_g, it updates the object and time value at this node and retraces the path back to the root node updating the object and time values on each of these nodes appropriately. The object heap of Fig. 12 results when we perform implicit update on the object heap of Fig. 11 at time t_1. The modified object heap shows that object O_2 is the closest object from time t_1 up to time t_2.

The above algorithm is accomplished by the recursive procedure $ImplicitUpdate(x)$ given below. This procedure acts as follows. If x is a cross node and O_1, O_2 are objects stored in its two children and $x.object = O_1$ then it sets $x.object$ to O_2 and sets $x.time$ to be the minimal of the times of its children and returns. Otherwise, it recursively invokes the procedure on a child y such that $y.time = x.time$. This recursive invocation may change the object and time at node y. Thus, when invocation on y returns, it reevaluates which of the objects at its children is the closest object and sets $x.object$ to that object and resets $x.time$ and returns. Note that the actual implicit update is carried out by invoking this procedure with the root r at time $r.time$.

```
ImplicitUpdate(x)
    y := x.left; z := x.right;
    If x.time<y.time and x.time<z.time (//x is a cross node)
        If x.object = y.object
            x.object := z.object;
        Else x.object := y.object;
        x.time := NextTime(y,z,x.time);
        return;
    If x.time = y.time
        ImplicitUpdate(y);
    If x.time = z.time
        ImplicitUpdate(z);
    x.object := CloserObject(y,z,x.time);
    x.time := min(y.time, z.time, NextTime(y,z,x.time));
    return
```

Note that for an object heap at time t, with root r, $t < r.time$, $r.time$ is less than or equal to $x.time$ for every node x in it, and the structure satisfies the heap property until $r.time$. At time $r.time$, we invoke the procedure $ImplicitUpdate(r)$. It is fairly straightforward to show that after invocation of $ImplicitUpdate(r)$ at time $r.time$, the structure continues to be an object heap.

We make the assumption, called *distinct-sibling-times* assumption, which for every pair of internal nodes y, z, that

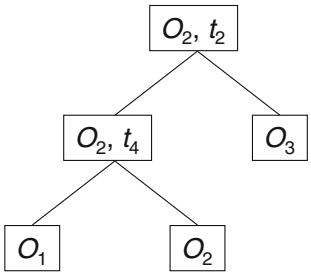

Fig. 12 The object heap resulting from an implicit update on the object heap of Fig. 11

are siblings, $y.time \neq z.time$. At the end of this subsection, we will discuss what happens when this assumption does not hold. Under the distinct-sibling-times assumption, when $ImplicitUpdate(r)$ is invoked, the algorithm will travel along a single path in the object heap. Hence, the complexity of executing $ImplicitUpdate(r)$ is $O(h)$ where h is the height of the object heap. Since $h \leq \log n + 1$, we see that the complexity of execution of the ImplicitUpdate is $O(\log n)$. Furthermore, at any point in time, there is only one event to monitor which is the expiration of the root node. Now the following theorem shows that there can be at most $O(n \log n)$ implicit updates, i.e., after at most $n \log n$ updates, $r.time = \infty$ and hence the object heap does not need any further implicit updates. Thus, the object heap is efficient in the sense that the number of implicit updates is only $\log n$ larger than the maximum number of times that the answer set may change.

Lemma 4.1 *Let T be an object heap at time 0 with root r, then after at most 2nlogn updates, r.time $= \infty$.*

Proof Observe that whenever an implicit update is performed, the object heap is traversed starting from the root r until a cross node x is reached. At the cross node x, the value of $x.object$ and $x.time$ is updated. It should be easy to see that the time when this update is carried out, i.e., at this time $x.time = r.time$, and at this time the closest object to O_q among $ObjectSet(x)$ changes and hence it is a split point for this set of objects. Since there are at most $2n_1 - 1$ split points for $ObjectSet(x)$ where n_1 is the number of objects in $ObjectSet(x)$, we see that the number of implicit updates on T that stop at x is bounded by $2n_1 - 1$. Let l be the level number of x. Now the total number of implicit updates that stop at a node at level l is bounded by twice the number of objects stored in the sub-trees whose root is at level l. Since the number of all such objects is the total number of objects, we see that the total number of implicit updates that stop at a level l node is bounded by $2n$. Since there are $\log n$ levels, we see that the total number of implicit updates is bounded by $2n \log n$. Clearly, after at most such implicit updates $r.time = \infty$. □

Since there are $O(n\log n)$ implicit updates and each such update takes time $O(\log n)$, we see that the cumulative complexity of performing these updates is $O(n\log^2 n)$.

If the distinct-sibling-times assumption does not hold then an invocation of $ImplicitUpdate(r)$ may travel on multiple paths, and hence the complexity may be higher than $O(h)$; however, its complexity can be shown to be only $O(n)$. Furthermore, the cumulative complexity over a number of invocations until $r.time = \infty$, can easily be shown to be still $O(n\log^2 n)$.

4.2.4 Explicit updates and piecewise linear model

Now we show how explicit updates can be implemented efficiently using object heap. We consider three types of updates, namely insertion of a new object, deletion of an existing object, and velocity-vector change of an existing object.

Addition. Assume that we have an object heap H with root node r storing n objects. In this heap which has $2n-1$ nodes, each leaf node will have a sibling. Assume that the new object to be inserted is O' at time *Ctime* which is the current time. We can assume that $Ctime < r.time$. We allocate two new nodes, say y_1 and y_2. Let x be the first leaf node in H, i.e., the leftmost leaf node at the lowest numbered level (note that if the height of H is h, then there can be leaf nodes both at level h-1 and h). Also let O'' be the object in node x. Now we add the two nodes y_1, y_2 as children of x and place objects $O'$and O''in these two nodes. This makes x as an internal node. Now we perform, the following operation, called $float(z)$ with argument x. This operation $float(z)$, is a recursive operation, which acts as follows. It sets *z.object* to the object among its children that is closest to the query object O_q at time *Ctime*, i.e., to the object given by the function $CloserObject(O_1, O_2, Ctime)$ and sets *z.time* to be the value given by the function $NextTime(O_1, O_2, Ctime)$ where O_1, O_2 are the objects *z.left.object* and *z.right.object*, respectively. After this, if z is the root then it stops; else it recursively invokes *float* on the parent of z. It is not difficult to see that the resulting structure H is an object heap at time *Ctime* containing the $n+1$ objects. It is complexity is clearly $O(\log n)$.

Consider the object heap of Fig. 11. Assume that $t_1 > 1$. When we insert a new object O_4 at time 1, the following heap results (see Fig. 13). Note that O_1 is still the closest object till t_1. Also O_3 is the closer of O_3 and O_4 until t_5. Here t_5 is assumed to be greater than t_1.

Deletion. Now we describe the deletion of an object from H. Assume that the object to be deleted is the object in the last leaf node, i.e. the right most node in the lowest level (i.e., at level h). Let y_2 be this node and let its sibling be y_1 and let x be their parent. It is not difficult to see that $x$is the last

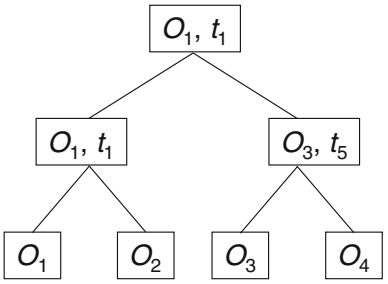

Fig. 13 The object heap resulting from an explicit update on the object heap of Fig. 11

internal node of H. We delete both the nodes y_1 and y_2 and place the object in y_1 in the node x. By this, H loses two nodes and x becomes a leaf node. Now, we invoke *float* operation on the parent z of x. After this, the resulting structure will satisfy the object heap property. Clearly, the complexity of this operation is $O(\log n)$.

Now consider the deletion of an object O' in an arbitrary leaf node y. To do this, we first delete the object O'' in the last leaf node of H using the above procedure. In the resulting object heap, let y' be the leaf node containing object O'. Notice that it is possible that y' is different from y. This occurs if y was the last but one leaf node in H. In any case, we replace object O' in y' by O'' and perform the *float* operation starting from the parent of y'. It should be easy to see that the resulting structure is an object heap containing all objects of H excepting O'. Clearly, the complexity of this whole procedure is $O(\log n)$.

Velocity-vector Change. When the speed and/or the direction of an existing object O′ changes, the square-distance function $d^2_{O'}(t)$ changes. In this case, find the leaf node y that contains O' and update $d^2_{O'}(t)$. After this, we perform the *float* operation starting from the parent of y. The complexity of this update is also $O(\log n)$.

Now we compare object heap and two existing algorithms in terms of their scalability to database size n. The two existing algorithms are kinetic tournament and off-line processing. Recall from Sect. 4.2.1 that the complexity of an update in kinetic tournament is $O(\log^2 n)$. Recall from Sect. 4.1 that the complexity of off-line processing $O(n\log n)$. This is also the complexity of off-line processing to process an update because for each update off-line processing has to re-compute all the answer-pairs. On the other hand, the above analysis shows that the complexity of an update in object heap is $O(\log n)$. The implication of this is that for a very large database, object heap is better by a factor of $\log n$, compared to kinetic tournament, in the worst case. Furthermore, object heap is better by a factor of n, compared to off-line processing, in the worst case. Thus object heap is more scalable to large databases.

Finally, let us consider the piecewise linear model in which each object can change its velocity-vector at most m times and we analyze the cumulative complexity of our on-line algorithm for maintaining the 1NN set and handling velocity-vector changes. The cumulative complexity is bounded as follows. Since the complexity for handling each velocity-vector update is $O(\log n)$, the total complexity of these updates is $O(nm\log n)$. Note that implicit updates are carried out between velocity-vector changes as before, as and when needed. Now, we bound the cumulative complexity of implicit updates. Observe that each object has a piecewise linear motion with at most m pieces throughout the time. Using Theorem 2.4, it is not difficult to see that the total number of implicit updates that stop at a particular internal node x is $O(n_1 m\alpha(n_1))$ where n_1 is the number of objects stored in the sub-tree rooted at x. Using the same argument as given in Lemma 4.1 for the linear case, we see that the total number of implicit updates that stop at an internal node at some level say l is $O(nm\alpha(n))$. Since there are at most $\log n$ levels, we see that the total number of implicit updates is $O(nm\alpha(n)\log n)$. Since the cost of each implicit update is $O(\log n)$, we see that the cumulative time complexity of implicit updates is $O(nm\alpha(n)\log^2 n)$. This is also the cumulative complexity of all the updates both implicit as well as explicit.

4.2.5 Extension to $k > 1$

For k $>$ 1, each internal node x in the object heap stores two items $x.set$ and $x.time$ such that $x.set$ is the set of the k closest objects to O_q up to time $x.time$ among all the objects in $ObjectSet(x)$. To achieve this goal, we modify the procedure $BuildObjectHeap(S, O_q, Ctime)$ as follows. For each internal node x, x.set is set to be the kNN set at time $Ctime$ among the objects stored at its two children; x.time is set to be the minimum of the time values in x's children and the next time when the kNN set among the objects stored at x's two children is going to change.

Now consider how to process an implicit update that is triggered at time t. Similar to $k = 1$ case, the implicit update algorithm traverses along a path of internal nodes $x_1, \ldots, x_g$ such that $x_1 = r$, x_g is a cross node and all nodes $x_1, \ldots, x_{g-1}$ are minimal nodes and the *time* value on all these nodes is $r.time$. After reaching x_g, it sets $x_g.set$ to be the kNN set at time t among the objects stored at x_g's two children and sets x_g.time to be the minimum of the time values in x_g's children and the next time when $x_g.set$ is going to change.

Explicit updates can be extended in the same fashion. It is not difficult to see that, when k is a constant, the complexity of our on-line algorithm for $k > 1$ is the same as that for $k = 1$.

5 Query processing with uncertainty

In this section, we discuss the processing of CPkNN queries. In Sect. 5.1, we discuss off-line processing and in Sect. 5.2, we discuss on-line processing.

5.1 Off-line processing

The off-line algorithm is based on Proposition 3.1 introduced in Sect. 3.1. According to this proposition, the PkNN set changes if and only if a min-curve intersects the max k-level. Thus, the algorithm first constructs the max k-level, then it finds the intersections between all the min-curves and the max k-level, and finally it determines the PkNN set created by each of these intersections. Specifically, the algorithm has the following steps:

Step 1: Compute the max k-level using the algorithm discussed in Sect. 4.1. The output of this step is the list of the edges in the max k-level sorted in ascending order of the time attribute of their start points.

Step 2: For each data object O_i, compute the intersections between O_i's min-curve and the max k-level. Call these intersections *critical intersections*. Denote the time coordinate of a critical intersection p by $p.time$. Since the edges in the max k-level are already sorted, the critical intersections of O_i are automatically sorted. For each intersection p of these intersections, compute D the first-order derivative of O_i's min-curve at p and D' the first-order derivative of the max k-level at p. If D is smaller than D', we say that p is an *entry point* and let $p.entry = i$. Otherwise, we say that p is an exit point and let $p.exit = i$. Intuitively, if p is an entry point, then O_i becomes a PkNN starting from $p.time$; if p is an exit point, then O_i stops being a PkNN starting from $p.time$.

Step 3: Sort the critical intersections of all the data objects in ascending order of their time attribute. Since the sequence of critical intersections is already sorted for each O_i, we use the multi-way merge algorithm introduced in [2] to merge these sequences to a global sequence. The algorithm uses a priority queue to store the smallest element and has complexity $O(M\log N)$ where N is the number of input sequences and M is the total number of elements in all input sequences.

Step 4: Find the set of objects whose min-curves are below the max k-level at time 0. Denote this set by S_1. Let $T_0 = 0$.

Step 5: For each critical intersection p_i, let $T_i = p_i.time$. Furthermore, if p_i is an entry point, then let $S_{i+1} =$

$S_i \cup \{p_i.entry\}$; otherwise, let $S_{i+1} = S_i - \{p_i.entry\}$.

Stpe 6: Let $T_h = \infty$ where h is the total number of critical intersections.

It is easy to see that the two output sequences given by $S_1 < S_2 < \cdots < S_h$ and $T_1, T_2 \cdots, T_h$ satisfy the property that $T_h = \infty$, and for each l, $1 \leq l \leq h$, S_l is the PkNN set during the interval $[T_{l-1}, T_l)$ and no two successive object sets in the S-sequence are identical.

Now we analyze the complexity of the above algorithm. In the linear model, the complexity of Step 1 is O($n$log$n$) as discussed in Sect. 4.1. The complexity of Step 2 is $\Theta(n^2)$ according to Corollary 3.3. The complexity of Step 3 is O(n^2log$n$) for multi-way merging. The complexity of Step 4 is n. The complexity of Step 5 is O(n^2). Thus, the complexity of the off-line algorithm in the linear model is O(n^2log$n$). Observe that the number of answer-pairs is $\Omega(n^2)$ (see Theorem 3.2), which gives a lower bound for the cost of any off-line algorithm. Our algorithm is only log$n$ higher than this lower bound.

In the piecewise linear model, the complexity of Step 1 is O($nm\alpha(n)$log$n$) as discussed in Sect. 4.1. The complexity of Step 2 is analyzed as follows. According to Theorem 2.4, there are O($nm\alpha(n)$) parabola pieces in the max k-level. Each min-curve has O(m) parabola pieces. According to Lemma 3.5, there are $O(nm\alpha(n))$ intersections between each min-curve and the max k-level. Thus, there are totally O($n^2m\alpha(n)$) intersections between n min-curves and the max k-level. The complexity of Step 3 is O($n^2m\alpha(n)$log$n$) for multi-way merging. The complexity of Step 4 is n. The complexity of Step 5 is O($n^2m\alpha(n)$). Thus, the complexity of the off-line algorithm in the piecewise linear model is $O(n^2m\alpha(n)\log n)$.

An alternative approach to off-line processing with uncertainty is a divide-and-conquer algorithm that is similar to the one discussed Sect. 4.1 for the certain case. It can be shown that the complexity of this approach is also O(n^2log$n$) for the linear model and $O(n^2 \cdot m \cdot \alpha(n) \cdot \log n)$ for the piecewise linear model.

5.2 On-line processing

For on-line processing, we need to track the change of the PkNN set. Recall that the PkNN set changes whenever a min-curve intersects the max k-level. Thus, we monitor two types of events. One is the change of the max k-level, called *max k-level events*, and another is critical intersections (i.e., the intersections between min-curves and the max k-level), called *intersection events*. At a high level, our on-line processing algorithm works as follows. We adapt the object heap data structure introduced in Sect. 4.2 to monitor max k-level events and use a priority queue to monitor intersection events. When an intersection event is triggered, an answer-pair is produced as an output. When a max k-level event is triggered, intersections between min-curves and the new max k-level are computed and the corresponding intersection events are scheduled. In Sect. 5.2.1, we present our on-line processing algorithm for k = 1 for the linear model. In Sect. 5.2.2, we extend this algorithm to the piecewise linear model. In Sect. 5.2.3, we discuss the extension to $k > 1$.

5.2.1 The linear model

In order to monitor max 1-level events, we adapt the object heap data structure as follows. We redefine the function $CloserObject(O_1, O_2, t)$ such that it returns the object that has the smallest maximum possible distance to O_q, among O_1 and O_2, at time t. We call the object heap constructed with this definition the *max object heap* and denote it by *MOH*. It is easy to see that if r is the root node of *MOH* then $r.object$ is the object that has the smallest maximum possible distance until the time $r.time - 1$.

Implicit updates to *MOH* are performed in the same way as to the object heap. Each implicit update causes either of the following two changes to the root node r: (i) both $r.object$ and $r.time$ change; and (ii) $r.object$ does not change but $r.time$ increases. Each implicit update, after its completion, triggers a max 1-level event. The max 1-level event is processed by the procedure Max1LevelUpdate(r, t) described below. Compute P the set of critical intersections that will occur between time t and $r.time$. For each critical intersection $p \in P$, create an intersection event $E =< p, p.time >$ which specifies that E is to be triggered at $p.time$ and it will cause a change to the P1NN set due to the critical intersection p. Build a priority queue of the created events such that the event with the earliest trigger time is the head. Call this priority queue the *event queue*.

The next event that is going to be triggered is either the max 1-level event, or the head of the event queue, whichever occurs earlier. When the i-th intersection event E is triggered, it is removed from the event queue and processed as follows. First, the P1NN set is updated. Specifically, let S_i be the P1NN set before the i-th event is triggered and let $E =< p, p.time >$. If p is an entry point (i.e., an object enters the answer set at $p.time$), then let $S_{i+1} = S_i \cup \{p.entry\}$. If p is an exit point (i.e., an object exits the answer set at $p.time$), then let $S_{i+1} = S_i - \{p.exit\}$. Second, let E' be the first event in the remaining event queue. If the remaining event queue is empty, let E' be the root of *MOH*. Output the pair $(S_{i+1}, E'.time)$ which indicates that the P1NN set is S_{i+1} from now on until E'.time. Notice that by the time the next implicit update occurs, the event queue will be empty.

The above process is repeated until r.time is ∞.

Now we study the complexity of our on-line processing algorithm. First, consider the complexity for processing

each individual event. The cost for processing an intersection event is O(log$n$) for removing the event from the event queue which has O(n) events. The cost for processing a max 1-level event involves executing the Max1LevelUpdate procedure. This procedure computes critical intersections and builds a priority queue of intersection events. There are O(n) critical intersections induced by each max 1-level event, and thus, the cost of the Max1LevelUpdate procedure is O(n) for computing O(n) critical intersections and building a priority queue of O(n) intersection events. Observe that there are only O($n$log$n$) max 1-level events in comparison with O(n^2) intersection events which only require log$n$ time each.

Now consider the total number of events that need to be processed. The total number of intersection events is $\Theta(n^2)$ according to Corollary 3.3. The total number of max 1-level events is O($n$log$n$) according to Lemma 4.1. Thus, the total number of events that need to be processed is O(n^2) which is of the same order as the number of times the P1NN set changes.

Finally, let us analyze the cumulative complexity of our on-line processing algorithm. The cumulative cost for processing intersection events is O(n^2log$n$). The cumulative cost for processing implicit updates is O($n$log^2n) as discussed in Sect. 4.2.3. The cumulative cost for processing max 1-level events is O(n^2). Thus, the cumulative complexity of our on-line processing algorithm is O(n^2log$n$). Observe that the number of answer-pairs is $\Omega(n^2)$ (see Theorem 3.2), which gives a lower bound for the cost of any on-line algorithm for maintaining the P1NN set. Our algorithm is higher by only a factor of log$n$ than this lower bound.

5.2.2 Explicit updates and piecewise linear model

The procedures for processing explicit updates in the piecewise linear model are described in Appendix D which is included in Electronic supplementary material. The following theorem (proved in Appendix D) gives the cumulative complexity of our on-line algorithm in the piecewise linear model.

Theorem 5.1 *Assume that each object can change its velocity at most m times. Then the cumulative complexity of the on-line algorithm for the uncertain case is* $O(n^2 m\alpha(n)\log n)$.

This is a good result in the following sense. According to Theorem 3.6, the number of answer-pairs is $\Omega(n^2 m)$, which is a lower bound for the cost of any algorithm for maintaining the P1NN set. Our on-line algorithm is only $\alpha(n)$log$n$ higher than this lower bound.

5.2.3 Extension to $k > 1$

For k $>$ 1, we want to use MOH to monitor the max k-level object. For this purpose, we modify MOH as follows. Each internal node x in MOH stores three items $x.kth$, $x.set$, and $x.time$, such that $x.kth$ is the object that has the k-th smallest maximum possible distance up to time $x.time1$ among all the objects in $ObjectSet(x)$ and $x.set$ is the set of the k objects that have the smallest maximum possible distances up to time $x.time1$ among all the objects in $ObjectSet(x)$. The purpose of storing $x.set$ is to enable the re-computation of $x.kth$ when an implicit update occurs.

It is not difficult to see that, when k is a constant, the complexity of our on-line algorithm for $k > 1$ is the same as that for $k = 1$.

6 Evaluation by experiments

In this section, we evaluate our work by experiments using real-world data. In the evaluation, we compare object heap with two existing algorithms namely kinetic tournament [5] and off-line processing [8]. We also compare the certain case with the uncertain case. The analysis conducted in previous sections compared these algorithms/cases in terms of the asymptotic worst-case complexity. Experiments help us understand in practice how they compare with each other. The performance measure is the cumulative processing time in a course during which the database receives a sequence of updates. All the compared algorithms are implemented in Java and executed on a Dell PowerEdge C6145 server with 48GB memory, 16-core AMD Opteron Magny-Cours 6136@2.4GHz. The operating system is Red Hat Enterprise Linux 6.

6.1 Test data

The test data were provided by Shanghai Jiaotong University. The data contain GPS traces collected from over 4,000 taxis running in the Shanghai urban area for 28 days [21]. Each GPS trace is a sequence of GPS points where each GPS point is a triple (x, y, t) which indicates that a taxi is at location (x, y) at time t. The difference in time between two consecutive GPS points is 1 minute. 4,000 taxis are too few for evaluation of scalability. In order to create a large-scale database, we split the GPS trace of each taxi into segments so that the time-length of each segment is 1 h. Then, for each 1-h segment, we created a moving object to travel it. Furthermore, all the moving objects were temporally aligned so that they start to move at the same time (time 0). For each of the 28 days and each taxi, we created moving objects using the 9-h period from 8 a.m. to 5 p.m. In this way, we created a pool of 1,087,893 moving objects where each object moves for 3,600 s starting from time 0.

Each GPS point is treated as a velocity-vector update. A taxi is assumed to move linearly at a constant speed between two consecutive GPS points. On average, each moving object

Table 2 Experimental parameters and their values

Parameter	Unit	Value
Number of Objects (N)		100, 1,000, 10,000, 1,00,000, 1,000,000
k		1–50 with increment of 5
Uncertainty region radius (r)	Meter	1, 5, 10, 15, 20, 25, 30
Simulated time	Second	3,600

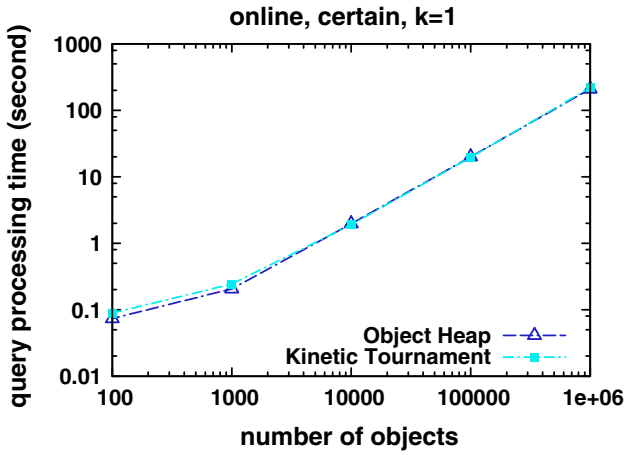

Fig. 14 Query processing time versus number of objects

generates one velocity-vector update per minute. For the uncertain case, the uncertainty region radius is a parameter which ranges from 1 to 30 meters. Indeed, the accuracy of a typical GPS receiver nowadays is < 15 m for 95 % of time (see e.g., [32]). With GPS augmentation technologies such as EGNOS and WAAS which are available in market, the positioning accuracy can be improved to < 3 meters for 95 % of time (see [32,33]). For an experiment run, N objects are randomly picked up from the pool. N is a system parameter ranging from 100 to 10,00,000. A random object is selected as the query object.

All the system parameters and their values are listed in Table 2.

6.2 Experimental results

In 6.2.1, we compare object heap and kinetic tournament. In 6.2.2, we compare on-line processing (object heap) and off-line processing. In 6.2.3, we compare the certain case and the uncertain case. In 6.2.4, we examine the space overhead of query processing (Fig.14).

6.2.1 Comparison between object heap and kinetic tournament

In Figure 15, the query processing time of the two algorithms as a function of k when $N = 10{,}000$. Here, the query processing time includes the time for initialization of the data structures, for processing of implicit updates as well as velocity-vector updates. From the figure, it can be seen that the query processing time is almost the same for object heap and kinetic tournament. On the other hand, the worst-case complexity of kinetic tournament is higher than that of object heap by a factor of $\log(N)$ (see Sect. 4.2). It is likely that a more skewed distribution of moving objects in space will reveal the theoretical gap in the experiments, making object heap more efficient. This conjecture is supported by an analysis in [2] that gives the average height of percolation in heap insertion.

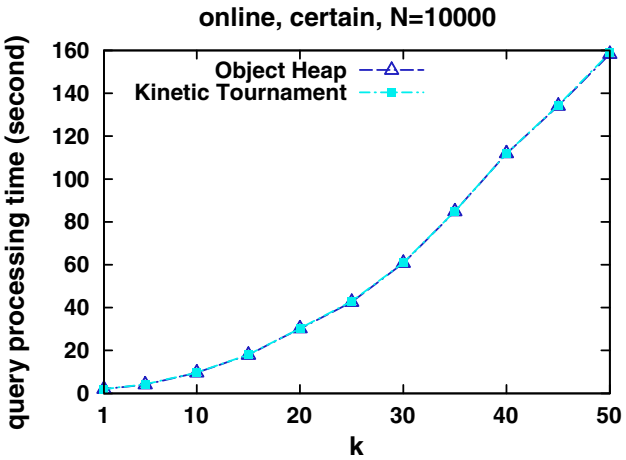

Fig. 15 Query processing time versus k

6.2.2 Comparison between on-line processing and off-line processing

First let us consider the scenario in which there are no velocity-vector updates. In this case, off-line processing and on-line processing compute the same number of answer-pairs. In order to create this scenario we used only the first two GPS points of each moving object. We executed off-line processing and on-line processing to compute the answer-pairs for the period of time starting from 0 until infinity. For off-line processing, we used the divide-and-conquer algorithm introduced in [8]. For on-line processing, we used object heap. The processing time of the off-line algorithm includes the time for one-time execution of the algorithm at time 0. The processing time of the on-line algorithm includes the time for initialization of the object heap and processing of implicit updates. Figure 16 shows the comparison result in the certain case. From the figure, it can be seen that the cost of on-line processing is higher than that of off-line processing by two orders of magnitude. However, the absolute difference remains small (up to 10 s). Indeed, when there are no updates there is no advantage of doing on-line processing.

Now we consider the case in which there are velocity-vector updates. For both off-line processing and on-line

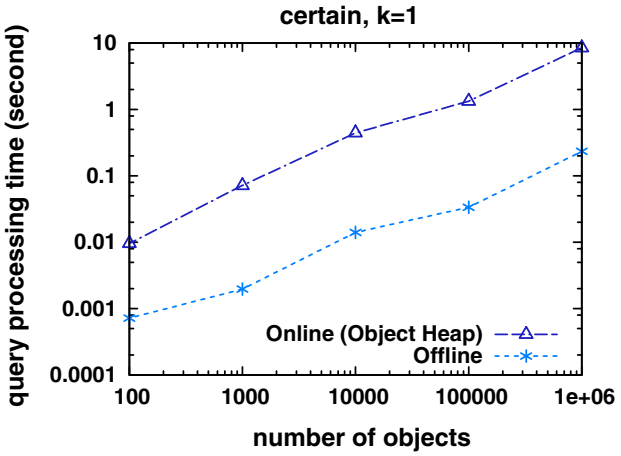

Fig. 16 Comparison between on-line and off-line as a function of the number of objects when there are no updates in the certain case

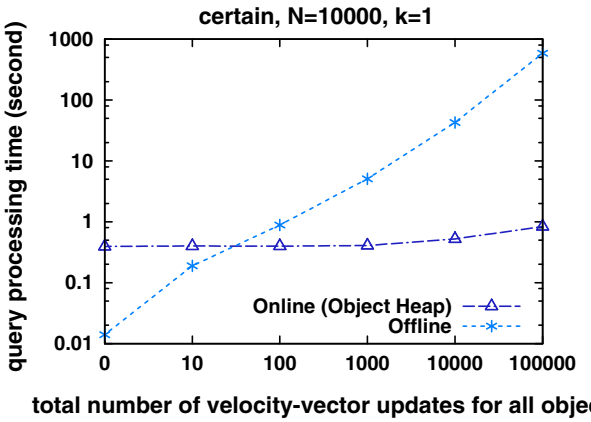

Fig. 17 Comparison between on-line and off-line as a function of the total number of velocity-vector updates for all the objects in the certain case

processing, we assumed that an object moves linearly to infinity after its last velocity-vector update. In this case, when a velocity-vector update occurs, off-line processing is executed to re-compute the answer-pairs for the period of time starting from current time until infinity. Thus, the processing time of the off-line algorithm includes the cumulative time for executions of the algorithm upon each velocity-vector update. The processing time of the on-line algorithm includes the time for initialization of the object heap, processing of implicit updates as well as velocity-vector updates (i.e., explicit updates). Figure 17 shows the comparison as a function of the total number of velocity-vector updates for all the objects in the certain case. From the figure, it can be seen that when there are fewer than 30 updates, off-line is better than on-line. However, when there are many updates, on-line is much better than off-line. When the number of updates reaches 1,00,000, the cost of off-line is one thousand times higher than that of on-line. Intuitively, if there

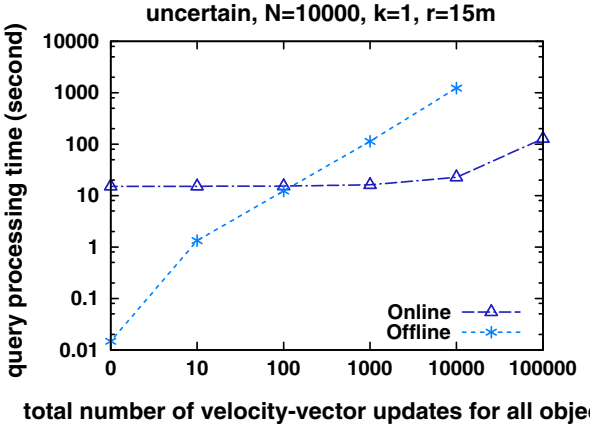

Fig. 18 Comparison between on-line and off-line as a function of the total number of velocity-vector updates for all the object in the uncertain case

are many updates, then off-line processing totally computes more answer-pairs than on-line processing and thus costs more time.

Figure 18 shows the comparison in the uncertain case. On-line processing starts to outperform off-line processing when the number of updates is higher than 100. Notice that there are totally 600,000 updates for all the objects when N = 10,000, whereas the maximum number of updates in Figs. 17 and 18 is 1,00,000. Thus, on-line is clearly much better than offline for practical usage, as the trend in Figs. 17 and 18 shows.

6.2.3 Comparison between certain case and uncertain case

In this subsection, we compare the certain case and the uncertain case in terms of the number of answer-pairs, the number of objects returned by each answer-pair. The comparison in terms of the query processing time is provided in Appendix G.

Number of Answer-pairs. Figures 19, 20 and 21 show the results for the number of answer-pairs. The data point for $N = 1{,}000{,}000$ for the uncertain case in Fig. 19 is not attainable due to a prohibitively long computation time. From the figures, it can be seen that the uncertain case has much more answer-pairs than the certain case. Furthermore, the difference between the two cases increases with the number of objects, k, and the uncertainty region radius. The huge difference between the certain case and the uncertain case matches the analytical result (recall the $O(n)$ factor of difference according to Theorems 2.4 and 3.4).

Size of PkNN Set. Now we study the size of the PkNN set which is the number of objects included in an answer-pair in the uncertain case. Figures 22, 23 and 24 show the size of the PkNN set as a function of the number of objects, k, and the

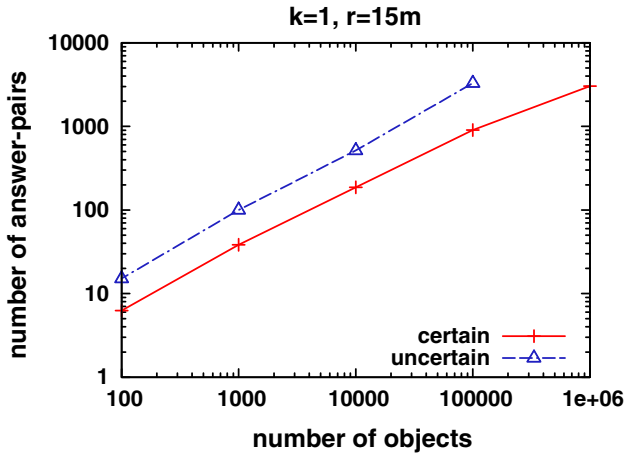

Fig. 19 Number of answer-pairs versus number of objects

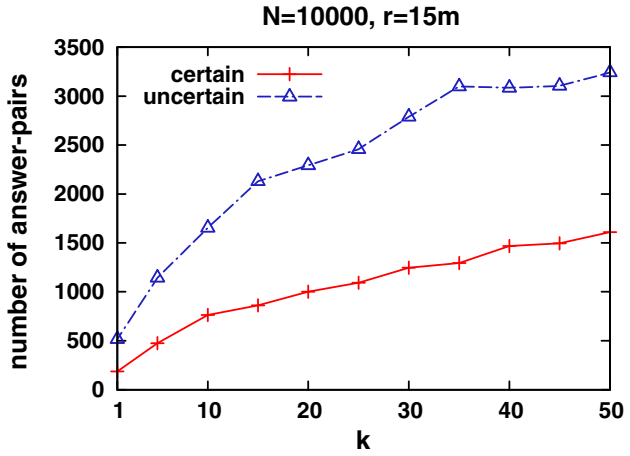

Fig. 20 Number of answer-pairs versus k

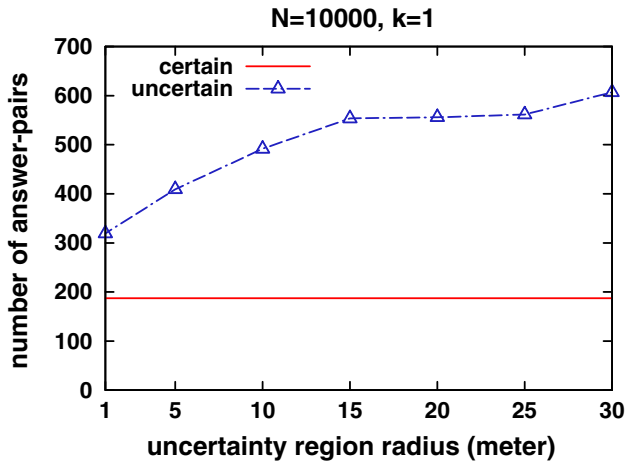

Fig. 21 Number of answer-pairs versus uncertainty region radius

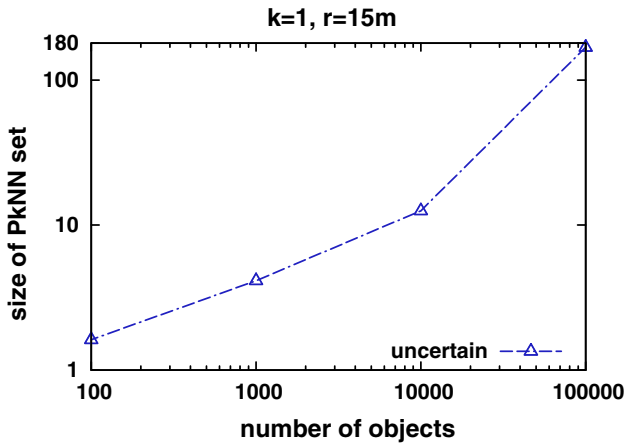

Fig. 22 Size of PkNN set versus number of objects

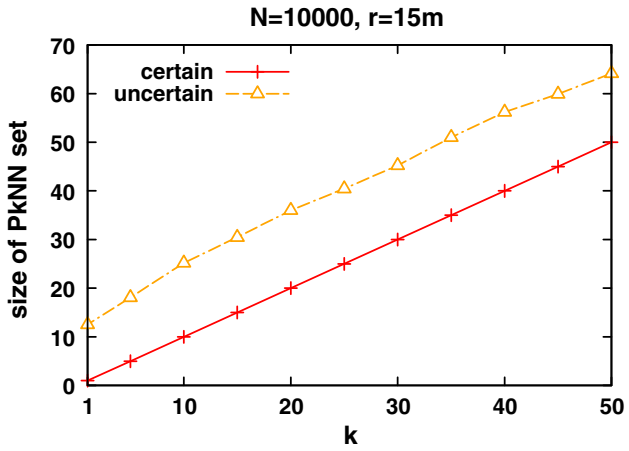

Fig. 23 Size of PkNN set versus k

uncertainty region radius, respectively. From these figures, it can be seen that the size of the PkNN set is bigger than k. Furthermore, this size increases with the number of objects, k, and the uncertainty region radius. Figure 23 compares the average size of an answer set returned by the uncertain case and that by the certain case. The figure shows that the uncertain case returns about 12 more objects than the certain case when there are 10,000 objects and the uncertainty region radius is 15 m.

6.2.4 Space overhead

We focus on the space overhead of on-line processing. First let us consider the space overhead in the certain case. The space overhead of object heap comes from the binary tree. Figure 25 shows the space overhead of object heap as a function of the number of objects for $k = 1$ and $k = 50$ respectively. From the figure it can be seen that the space overhead is linear in the number of objects. This is because the number of nodes in the binary tree is linear in the number of objects.

Observe that when there are 1 million objects, the space overhead reaches 800 MB when $k = 1$. This suggests that object heap is well suitable for in-memory processing. The space overhead can be reduced by storing the binary tree in an array

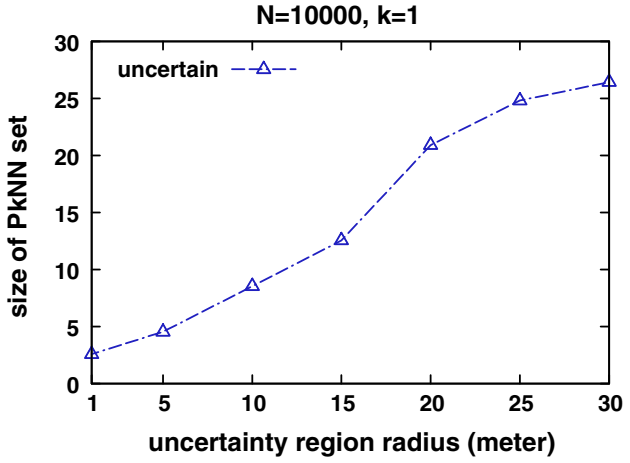

Fig. 24 Size of PkNN set versus uncertainty region radius

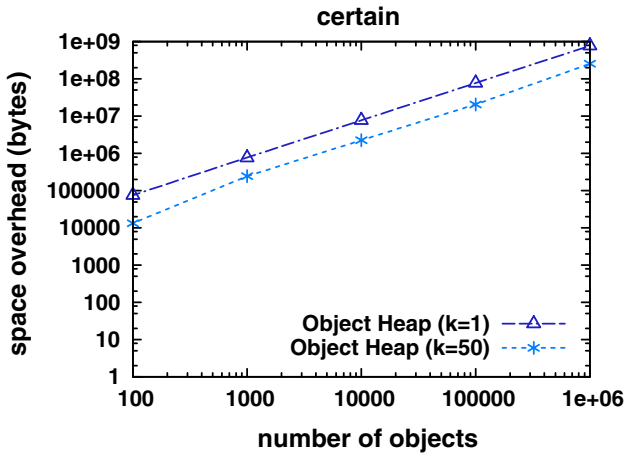

Fig. 25 Space overhead of object heap versus number of objects

and thus getting rid of pointers. However, space optimization is not the focus of this work.

Let us examine the space overhead for different values of k. One would think that the space overhead should increase with k since the size of the kNN set stored at each node of the binary tree increases with k (see Sect. 4.2.5). However, Fig. 25 shows that the space overhead for $k = 50$ is actually smaller than that for $k = 1$. The reason for this is as follows. Notice that each leaf node in the binary tree stores at most k objects. It is not difficult to see that the memory overhead of the binary tree is

$$\left(2 \cdot \left\lceil \frac{N}{k} \right\rceil - 1\right)(H + k \cdot c) \approx 2 \cdot N \cdot \left(\frac{H}{k} + c\right) \qquad (1)$$

where H is the fixed space overhead of each node excluding the overhead of the kNN set, c is the space overhead for each member of the kNN set. From Eq. (1), it can be seen that the space overhead decreases as k increases.

Now we study the space overhead in the uncertain case. Compared with the certain case, the only additional data structure used by the uncertain case is the event queue (see Sect. 5.2.1). We traced the average length of the event queue and the results are shown in Table 3. From Table 3, it can be seen that the additional space overhead introduced by the uncertain case is negligible.

Table 3 Average length of the event queue in the uncertain case when k = 1 and r = 15 m

Number of objects (N)	100	1,000	10,000	1,00,000
Avg. length of event queue	1.0	1.0	1.2	1.3

6.3 Experiments with raw trajectories

For the results that have been presented so far, we used hourly trajectories that are split from raw trajectories. In order to verify whether these results carry over to raw trajectories, we conducted experiments with raw trajectories. Recall that we have 4,000 raw trajectories for each of the 28 days and each raw trajectory is 9 h long. Thus, the maximum number of moving objects that can be generated from raw trajectories without splitting is $4{,}000 \times 28 = 1{,}12{,}000$. We conducted experiments with the number of moving objects varying from 100 to 1,00,000. When N is smaller than 4,000, we used the raw trajectories in the first day. In this case, there is no realignment or splitting. When N is bigger than 4,000, we used raw trajectories from multiple days and realigned them to start at the same time. In this case, there is realignment but no splitting.

In terms of performance measures, we focused on the query processing time and the number of answer-pairs since these two are studied in our theoretical analysis. Observe that the results for these measures depend on the time-length of trajectories. In order for the results with split trajectories (1 h each) and those with raw trajectories (9 h each) to be comparable, we adjusted the two measures so that they are insensitive to the trajectory time length. Particularly, we define *normalized processing time* to be the query processing time divided by the trajectory time length. Thus, the normalized processing time represents the amount of time required to process the query during each second of the trajectory. We define *answer-pair density* to be the number of answer-pairs divided by the trajectory time length. Thus, the answer-pair density represents the number of answer-pairs during each second of the trajectory.

Figures 6.16 and 6.17 (see Appendix E which is included in Electronic supplementary material) show the comparison in terms of normalized processing time and answer-pair density, respectively. The parameter configurations of the two figures are the same as those of Figs. 25 and 19, respectively. From Figures 6.16 and 6.17, it can be seen that in most cases the results with raw trajectories are fairly close to the results

with split trajectories. Notice that when N is small (100, 1,000), there is a big gap in terms of normalized processing time between raw trajectories and split trajectories (see Figure 6.16). Specifically, the normalized processing time for raw trajectories is smaller than that for raw trajectories. This is because the initialization cost, which is fixed regardless of the trajectory time length, occupies a significant portion of the overall processing time when N is small. Thus, for raw trajectories, the initialization cost is amotized over a longer period of time. These results show that our approach of splitting trajectories to create a much bigger pool is justified.

7 Conclusion and future work

The theoretical results developed in this paper are summarized in Table 1 in Sect. 1. Furthermore, we evaluated the proposed algorithms by experiments using real-world GPS traces. The results showed that (i) even though object heap is better than kinetic tournament by a factor of $\log(n)$ in the worst case, it is equally efficient as the latter one for the data set we tested, in the average case; (ii) without any indexing structure, query processing easily scales to 1 million objects in the certain case and it scales to 10,000 objects in the uncertain case; (iii) on-line processing is much more efficient than off-line processing for practical update rates; and (iv) the space overhead of query processing fits in the main memory.

Correlation. We envision the correlation between moving objects to be considered in preparing the trajectories for *knn* computation. In this sense, the issue of correlation is related to the model rather than to the *knn* computation. As we indicated in the introduction, the piecewise linear motion model and the circular uncertainty-region model are widely accepted in the literature. The question is how to incorporate correlation in these models, and in our view it is orthogonal to the query processing issue we study in this paper. Assuming that this question is resolved, the models of trajectories, which now reflect correlation, are processed by the methodology we propose here. In other words, we envision that the correlation among moving objects is incorporated at the stage when the trajectories of moving objects are constructed. After this construction takes place, the trajectories can be input to our algorithms for query processing.

Now, how to incorporate correlation in the modeling of trajectories? This is a rich research problem that has been studied previously (see, e.g., [37]), although many facets are still open. In [37] the authors propose the following way. Given two consecutive locations (x_1, y_1) and (x_2, y_2) of a moving object o, we construct a piecewise linear trajectory of o between the two. The route of the trajectory is constructed using a shortest path between (x_1, y_1) and (x_2, y_2). The speed of each linear piece, and therefore the time of arrival at each linear endpoint, is based on the traffic conditions determined from other objects that are tracked by the database server, i.e., by correlating moving objects.

Finally, there are many situations in which there is no correlation among moving objects. In one case, objects naturally move independently of each other. Examples include pedestrians, vehicles on uncongested roads, and non-herding animals such as tigers. In another case, the objects tracked by the database server are chosen to be the ones that move independently. For example, consider the situation in which we track groups of people. In this situation, we choose a representative from each group to track and these representatives move independently.

Other uncertainty models. In this paper, we assume that the uncertainty region of each object is a circle with a fixed radius. Other uncertainty models are possible. First of all, the radius may change per time, e.g., determined by the product of the maximum speed of the object and the time since the last location update (see [1]). For objects that move along straight line paths, the uncertainty region is a line-segment (see [11]). It is also possible that both the speed and the direction of the object change within a certain range. In this case, the uncertainty region is a fan area (see [12]). For these uncertainty models, many principles used in this paper, such as Proposition 3.1 for determining the kNN set, is still applicable. However, the shapes of max-curves and min-curves are different for different uncertainty models.

Another uncertainty model allows that there is location uncertainty associated with the query object as well. This happens, for example, when the query processing is performed at a central database, where the location is imprecise for all objects including O_q. In this case, Proposition 3.1 does not hold anymore. Specifically, O_i's min-curve being below the k-th level at a time t is only a necessary, but not a sufficient condition, for O_i to be a PkNN at t. For details see, Appendix F which is included in Electronic supplementary material. The implication of this observation is that when there is uncertainty associated with the query object the PkNN set cannot be determined just based on the maximum and minimum possible distances. Thus, the answer-pairs probably cannot be computed merely in the Time-Square_Distance space; the relationship among the objects in the motion space needs to be examined as well.

Other query semantics. In this paper, we define a query to ask for possible kNNs. Another useful query semantics would be to ask for the objects that are *definitively* the kNNs regardless of the location configuration. When $k = 1$, a "definitely" query can be easily processed as follows. First we compute the P1NN set (i.e., the set of objects that are possible 1NNs) using an algorithm introduced in this paper. If there is only one object in the P1NN set, then that object is

the answer to the "definitely" query; otherwise, the answer to the "definitely" query is empty. However, this approach does not straightforwardly extend to $k > 1$. If there are exactly k objects in the PkNN set, it is still true that these k objects are the answer to the "definitely" query. However, if there are more than k objects in the PkNN set, we cannot say that the answer to the "definitely" query is empty because k of these objects may be definitely the kNNs.

Indexing in the Time-Square_Distance space. In existing studies, indexes are built in the motion space (see [7,12]). Would it be more efficient if we build indexes in the Time-Square_Distance? That is, we index the distance curves using a spatial indexing structure such as a quadtree. A quick observation is that the distance curves are invariable in time unless there are updates. On the other hand, the locations of moving objects change continuously in the motion space even if there are no updates. Thus intuitively an index structure in the Time-Square_Distances space would be more stable than one in the motion space.

Acknowledgments This research was supported in part by the US Department of Transportation National University Rail Center (NURAIL); Illinois Department of Transportation (METSI); and National Science Foundation grants IIS-1213013, CCF-1216096, DGE-0549489, IIP-1315169, CCF-0916438, CNS-1035914, CCF-1319754, CNS-1314485.

References

1. Hornsby, K., Egenhofer, M.: Modeling moving objects over multiple granularities. Ann. Math. Artif. Intell. **36**(1–2), 177–194 (2002)
2. Weiss, M.A.: Data Structures and Algorithms Analysis in C++. Benjamin/Cummings, Reading (1994)
3. Li, Y., Yang, J., Han, J.: Continuous K-Nearest Neighbor Search for Moving Objects. SSDBM pp. 123–126 (2004)
4. Iwerks, G., Samet, H., Smith, K.: Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. VLDB (2003)
5. Basch, J., Guibas, L.J.: Data structures for mobile data. J. algorithm **31**, 1–28 (1999)
6. da Fonseca, G.D., de Figueiredo, C.M.H.: Kinetic heap-ordered trees: tight analysis and improved algorithms. Inf. Process. Lett. **85**(3), 165–169 (2003)
7. Huang, Y., Lee, C.: Efficient evaluation of continuous spatio-temporal queries on moving objects with uncertain velocity. Geoinformatica **14**, 163–200 (2010)
8. Agarwal, P.K., Sharir, M.: Davenport-schinzel sequences and their geometric applications. In: Sack, J.R., Urrutia, J. (eds.) Handbook of computational geometry. North-Holland, Amsterdam (2000)
9. Agarwal, P.K., Sharir, M.: Arrangements and Their Applications. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry. North-Holland, Amsterdam (2000)
10. Trajcevski, G., Tamassia, R., Ding, H., Scheuermann, P., Cruz, I.: Continuous probabilistic nearest-neighbor queries for uncertain trajectories. EDBT (2009)
11. Huang, Y., Hen, C., Lee, C.: Continuous K-nearest neighbor query for moving objects with uncertain velocity. Geoinformatica **13**, 1–25 (2009)
12. Huang, Y., Liao, S., Lee, C.: Evaluating continuous K-nearest neighbor query on moving objects with uncertainty. Inf. Syst. **34**, 415–437 (2009)
13. Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Querying imprecise data in moving object environments. TKDE **16**(9), 1112–1127 (2004)
14. Trajcevski, G., Tamassia, R., Cruz, I.F., Scheuermann, P., Hartglass, D., Zaimerowski, C.: Ranking continuous nearest neighbors for uncertain trajectories. VLDB J. **20**(5), 767–791 (2011)
15. Mokhtar, H., Su, J., Ibarra, O.: On Moving Object Queries. PODS (2002)
16. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. SSTD, (2001)
17. Kolahdouzan, M. Shahabi, C.: Continuous K-nearest neighbor queries in spatial network databases. In: STDBM (2004)
18. Xiong, X., Mokbel, M., Aref, W.: SEA-CNN: scalable processing of continuous K-nearest neighbor queries in spatio-temporal databases. In: ICDE (2005)
19. Sistla, A., Wolfson, O., Xu, B., Rishe, N.: Answer-pairs and processing of continuous nearest-neighbor queries. In: Proceedings of the 7th International Workshop on Foundations of Mobile Computing, San Jose, CA (2011)
20. Clarkson, K.L., Shor, P.W.: Applications of random sampling in computational geometry, II. Discrete Comput. Geom. **4**(1), 387–421 (1989)
21. SJU Traffic Information Grid Team, Grid Computing Center. Shanghai taxi trace data. http://wirelesslab.sjtu.edu.cn/download.html
22. Mokbel, M., Chow, C., Aref, W.G.: The new casper: query processing for location services without compromising privacy. In: VLDB (2006)
23. Wolfson, O., Jiang, L., Sistla, P., Chamberlain, S., Rishe, N., Deng, M.: Databases for tracking mobile units in real time. In: ICDT (1999)
24. Guting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann, Los Altos (2005)
25. Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W.G., Hambrusch, S.: Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects. IEEE Trans. Comput. **15**(10), 1124–1140 (2002)
26. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the positions of continuously moving objects. In: ACM SIGMOD (2000)
27. Chen, J., Cheng, R.: Efficient evaluation of imprecise location-dependent queries. In: ICDE (2007)
28. Cheng, R., Chen, L., Chen, J., Xie, X.: Evaluating probability threshold k-nearest-neighbor queries over uncertain data. In: EDBT (2009)
29. Sistla, P.A., Wolfson, O., Chamberlain, S., Dao, S.: Querying the uncertain position of moving objects. In: Temporal Databases: Research and Practice (1998)
30. Cheng, R., Kalashnikov, D., Prabhakar, S.: Evaluating probabilistic queries over imprecise data. In: ACM SIGMOD (2003)
31. Chen, J., Cheng, R., Mokbel, M., Chow, C.: Scalable processing of snapshot and continuous nearest-neighbor queries over one-dimensional uncertain data. The Very Large Database J. (VLDBJ) **18**, 1219–1240 (2009)
32. http://www8.garmin.com/products/gps60/spec.html
33. http://egnos-portal.gsa.europa.eu/
34. Becker, L., Blunck, H., Hinrichs, K., Vahrenhold, J.: A framework for representing moving objects. In: Proceedings of the 15th International Conference on Database and Expert Systems Applications (2004)
35. Forlizzi, L., Guting, R.H., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In: Proceedings

of the ACM SIGMOD International Conference on Management of Data (2000)
36. Xie, X., Yiu, M., Cheng, R., Lu, H.: Scalable evaluation of trajectory queries over imprecise location data. IEEE Trans. Knowl. Data Eng. (99). doi:10.1109/TKDE.2013.77
37. Trajcevski, G., Wolfson, O., Hinrichs, K., Chamberlain, S.: Managing uncertainty in moving objects databases. ACM Trans. database Syst. (TODS) **29**(3), 463–507 (2004)