Information Systems 35 (2010) 911–932

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Optimized algorithms for predictive range and KNN queries on moving objects

Rui Zhang [a,*], H.V. Jagadish [b], Bing Tian Dai [c], Kotagiri Ramamohanarao [a]

[a] Department of Computer Science and Software Engineering, University of Melbourne, Australia
[b] Department of Electrical Engineering and Computer Science, University of Michigan, United States
[c] Department of Computer Science, National University of Singapore, Singapore

ARTICLE INFO

Article history:
Received 10 February 2009
Received in revised form
13 May 2010
Accepted 16 May 2010
Recommended by F. Korn

Keywords:
Transformed Minkowski Sum
Spatio-temporal databases
Moving objects
Range query
Nearest neighbor query
kNN

ABSTRACT

There have been many studies on management of moving objects recently. Most of them try to optimize the performance of predictive window queries. However, not much attention is paid to two other important query types: the predictive range query and the predictive *k* nearest neighbor query. In this article, we focus on these two types of queries. The novelty of our work mainly lies in the introduction of the Transformed Minkowski Sum, which can be used to determine whether a moving bounding rectangle intersects a moving circular query region. This enables us to use the traditional tree traversal algorithms to perform range and kNN searches. We theoretically show that our algorithms based on the Transformed Minkowski Sum are optimal in terms of the number of tree node accesses. We also experimentally verify the effectiveness of our technique and show that our algorithms outperform alternative approaches.

1. Introduction

As telecommunication technologies such as GPS and mobile devices become widely used, we are able to track cars or mobile phone users and provide location-based services to them. Commonly, these services request information about moving objects (cars, mobile phone users, etc.) for a period of time in the future, which are called *predictive queries*. Several structures have been proposed to support efficient processing of predictive queries [19,24,13,16]. All these studies focus on one query type, the *predictive window query*, which is defined as follows. Given a rectangular region WQ and a future time interval Q_T, retrieve the set of objects that will intersect WQ at any timestamp $t \in Q_T$. For a real-life example, a police officer can issue the query: "report the registration numbers of all the cars that will pass through the central business district in the next 10 min" through a traffic monitoring system. A predictive window query is suitable here since the central business district is probably best described by a rectangle. However, in many other cases, a query region is best described by a circle. For example, a tourist can issue the query: "find all the vacant taxis within 200 m from me in the next 10 min" through a mobile phone. The region of this query is more accurately represented by a circle than by a rectangle. In this regard, we introduce the *predictive range query* to capture predictive queries with circular regions. A formal definition of the query will be given in Section 2.1.

Consider further the above example. Sometimes, the tourist may not know a proper distance to specify the query range. Suppose there is no taxi that will be within 200 m from the tourist in the next 10 min, then the tourist will get a null answer to the query, and not get any taxi. In this case, another type of query, the *predictive nearest neighbor query* suits the purpose better. The tourist would

* Corresponding author. Tel.: +61 3 83441345.
E-mail addresses: rui@csse.unimelb.edu.au (R. Zhang), jag@umich.edu (H.V. Jagadish), daibingt@comp.nus.edu.sg (B.T. Dai), rao@csse.unimelb.edu.au (K. Ramamohanarao).

0306-4379/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2010.05.004

now ask: "find the nearest vacant taxi to me in the next 10 min". Generally, the tourist can request k nearest taxis for consideration, which corresponds to a *predictive k nearest neighbor* (*kNN*) *query*. A formal definition of the query will be given in Section 2.2.

While the predictive range and kNN queries have wide ranges of applications, few studies have specifically focused on optimizing algorithms to process these queries. In this article, we study how to process these two types of queries efficiently and make the following contributions:

- We introduce an analysis tool, the *Transformed Minkowski Sum* (TMS), for moving object databases. TMS can be used to determine the intersection of two moving objects of arbitrary shapes. In particular, we apply this technique to determine the intersection between a moving rectangle and a moving circle, which enables us to process queries with circular search regions on moving objects, specifically, predictive range and kNN queries.
- Exploiting the TMS, we derive an equivalent condition that is easy to evaluate for identifying objects in a moving circular query range. Based on this equivalent condition, we can adopt a tree traversal algorithm for processing the predictive range query. The algorithm can return *exactly* the objects that intersect the circular range, which is not achieved by any existing algorithm. We prove that this algorithm is optimal in terms of the number of tree node accesses.
- We also provide tree traversal algorithms for the predictive kNN query, again, enabled by the TMS concept. We prove that our kNN algorithm is optimal in terms of the number of tree node accesses. In addition, we can determine the timestamps when the closest distances between the query point and the k nearest neighbors (NNs) happen.
- Based on the TMS, we develop a cost model to estimate the number of node accesses for predictive range queries. We also show how it can help estimate the cost of predictive kNN queries.
- We perform an extensive experimental study. The results verify the effectiveness of our algorithms and the accuracy of our cost model.

The rest of the article is organized as follows: Section 2 gives the formal definitions of the predictive range and kNN queries and discusses our problem setting. Section 3 reviews related work and provides preliminaries. In Section 4, we introduce the Transformed Minkowski Sum. Sections 5 and 6 present our algorithms for the predictive range and kNN queries, respectively. Section 8 reports the results of our experimental study and Section 9 concludes the article.

2. Problem formulation

In this section, we first give the formal definitions of the predictive range query and the predictive kNN query. Then we discuss our problem setting and optimization goals.

2.1. Predictive range queries

In previous studies on the predictive window query (such as [19,16]), three kinds of queries are distinguished based on the time span and the region they specify. Similarly, we distinguish three kinds of queries for the predictive range query: *timeslice query*, *time-interval query* and *moving query*. Let $\boldsymbol{S}(Q,r)$ denote a circle centered at point Q with radius r; let t_c be the current timestamp and t_{ref} be a reference timestamp. The definitions of the three kinds of predictive range queries are given below.

Definition 1 (*Timeslice range query*). Given a set of moving objects MS, a point Q, a radius r and a timestamp t ($t \geq t_c$), find every object $O \in MS$ that satisfies: O intersects $S(Q,r)$ at timestamp t.

Definition 2 (*Time-interval range query*). Given a set of moving objects MS, a point Q, a radius r and a time interval $Q_T = [t_{\vdash}, t_{\dashv}]$ ($t_c \leq t_{\vdash} \leq t_{\dashv}$), find every object $O \in MS$ that satisfies: there exists a timestamp $t \in Q_T$ so that O intersects $S(Q,r)$ at t.

In general, the query range may move and change its radius. For example, a tornado (which is a circular region) is moving and expanding. We would like to find all the cell phone users (moving objects) that may be contained in the tornado in the next half hour so that they can be contacted and notified of the danger. Bush fire is a natural disaster that happens every year in Australia. The range of the fire is an expanding circular region and the center of the fire moves due to wind. When there is a bush fire, it is urgent to notify anyone who may be caught in the fire in the next few hours. In case of an oil slick on the sea, the oil slick expands as a circular region and the center of the circle moves as the water flow. Vessels should be notified to avoid the oil slick regions Therefore, we provide the following more general form of the range query.

Definition 3 (*Moving range query*). Given a set of moving objects MS, a moving point Q represented by its position $Q(t_{ref})$ at the reference time t_{ref} and its velocity Q_V, a changing radius represented by its radius $r(t_{ref})$ at the reference time t_{ref} and its changing speed r_v, and a time interval $Q_T = [t_{\vdash}, t_{\dashv}]$ ($t_c \leq t_{\vdash} \leq t_{\dashv}$), find every object $O \in MS$ that satisfies: there exists a timestamp $t \in Q_T$ so that O intersects $S(Q(t),r(t))$ at timestamp t, where $Q(t)$ is the query point's position at timestamp t.

In Definition 3, $Q(t)=Q(t_{ref})+Q_V(t-t_{ref})$ and $r(t)=r(t_{ref})+r_v(t-t_{ref})$. Note that Q and Q_V are vectors while r and r_v are scalars. The time-interval range query is a special case of the moving range query: the timeslice range query is a special case of the time-interval range query. In the rest of the article, we also call Q_T the *querying period*.

Fig. 1 shows examples of the three kinds of predictive range queries in a 2-dimensional space. Together with the time dimension, the coordinate space is 3-dimensional.

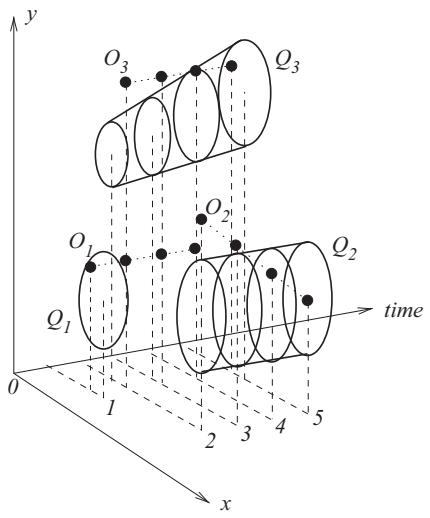

Fig. 1. Predictive range query examples.

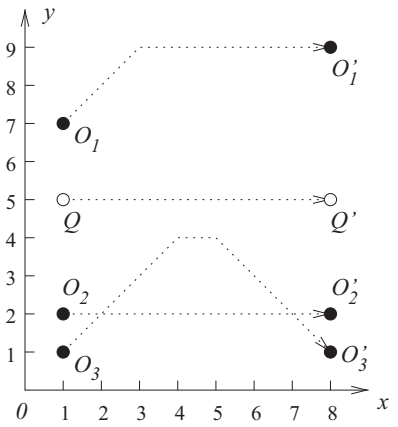

Fig. 2. Predictive kNN query examples.

We use point objects in these examples for ease of presentation (same for the examples in Fig. 2), although the following discussions also apply to objects with extents. Q_1 is a timeslice range query[1] at timestamp 1. Its query region is a disk. Object O_1 is in it, while O_2 or O_3 is not in it. Therefore, the answer to Q_1 is O_1. Q_2 is a time-interval range query spanning the period [2,5]. Its query region is a cylinder. At timestamp 2, no object is in Q_2. Object O_2 is moving and it moves into Q_2 at timestamps 4 and 5. Objects O_1 and O_3 are not moving and they stay outside of Q_2 all the time. Therefore, the answer to Q_2 is O_2. Q_3 is a moving range query spanning the period [2,5]. The center and radius of Q_3 are both changing during the querying period. The query region of Q_3 is a leaning truncated cone. No object is in Q_3 at timestamp 2. Although O_3 does not move, it is in Q_3 at timestamps 4 and 5 because of the movement of Q_3. The other two objects are outside of Q_3 all the time. Therefore, the answer to Q_3 is O_3. From these examples, we can see that the relative movements of objects and time are important factors to determine answers.

[1] We misuse Q to also denote a predictive range or kNN query (rather than just a query point) when the context is clear.

2.2. Predictive K nearest neighbor queries

Similar to the predictive range queries, we have three kinds of predictive k nearest neighbor queries: timeslice, time-interval and moving queries; the timeslice query is a special case of the time-interval query and the time-interval query is a special case of the moving query. In the following, we only give the definition of the most general version for brevity. Let $closest(O,Q,T)$ denote the closest distance between object O and point Q during a time interval T.

Definition 4 (*Moving k nearest neighbor (kNN) query*). Given a set of moving objects MS, a moving point Q represented by its position $Q(t_{ref})$ at the reference time t_{ref} and its velocity Q_V, an integer k and a time interval $Q_T=[t_{\vdash},t_{\dashv}]$ $(t_c \le t_{\vdash} \le t_{\dashv})$, find a subset A of MS that satisfies: (1) $|A|=k$; (2) $\forall O \in A$ and $\forall O' \in MS-A$, $closest(O,Q,Q_T) \le closest(O',Q,Q_T)$.

Fig. 2 shows an example of the moving kNN query in a 2-dimensional space: O_1, O_2, O_3 are moving objects and Q is the moving query point. The time axis is not shown in this figure for a clearer representation of the movements of the objects; the time for each position of an object can be easily derived from the object's trajectory. The dotted lines show the trajectories of the objects' movements from timestamp 1 to timestamp 8 (arrows showing the directions); O_1', O_2', O_3' and Q' are the final positions of the objects and query, respectively. All the objects and the query point have the same speed, 1, in dimension x for the whole period [1,8]. Their speeds in dimension y can be derived from their trajectories. During the movement of O_1, it becomes closest to Q at timestamp 1, and the closest distance is 2, so $closest(O_1, Q, [1,8])=2$. Similarly, we have $closest(O_2, Q, [1,8])=3$ and $closest(O_3, Q, [1,8])=1$. Therefore, during timestamp [1,8], the moving NN to Q is $\{O_3\}$ and the moving 2NN to Q is $\{O_3, O_1\}$.

We should distinguish the moving NN query from the continuous NN query. The continuous NN query returns the NN of a query point at every timestamp of the querying period, that is, answering the query continuously. Suppose we want to answer the continuous NN query for Q in Fig. 2 for the period [1,8]. As O_1 is the NN in time interval [1,2], O_3 is the NN in the period (2, 7], and O_2 is the NN in the period (7,8], the answer for the continuous NN query is $\{\langle O_1,[1,2]\rangle, \langle O_3,(2,7]\rangle, \langle O_2,(7,8]\rangle\}$. The answer of a continuous NN query is a set of pairs $\{\langle A_i,T_i\rangle\}$, where T_i is the time interval when A_i is the NN of the query. Obviously $\cup_i T_i = Q_T$ and $i \ne j \Rightarrow T_i \cap T_j = \emptyset$. The above discussion is with regard to one NN. The extension to kNN is straightforward.

Now we can see the difference between the predictive kNN query and the continuous kNN query. The former requests one answer set for the whole querying period

while the latter requests an answer set at every timestamp in the querying period. Both query types have their applications. For example, suppose there is a swarm of robots moving around. A working robot needs maintenance within the next 10 min, so it asks: "which maintenance robot comes closest to me within the next 10 min?" This is a predictive kNN query because finding one maintenance robot (the nearest one within the next 10 min) is enough. In another scenario, suppose there are a number of tanks moving around in a battlefield. The driver of a tank issues the query: "report to me the nearest three enemy tanks at any time for the next 10 min". This is a continuous kNN query because the driver should be alerted with the nearest enemy tanks all the time.

We can derive the answer of a predictive kNN query from the answer of its continuous kNN query counterpart. In the example shown in Fig. 2, given the answer for the continuous NN query, $\{\langle O_1,[1,2]\rangle, \langle O_3,(2,7]\rangle, \langle O_2,(7,8]\rangle\}$, we just need to compare $closest(O_1,Q,[1,2])$, $closest(O_3,Q,(2,7])$ and $closest(O_2, Q, (7,8])$ and get the closest one, which is O_3. However, maintaining kNN for all timestamps is much less efficient and unnecessary for the predictive kNN query. In this article, we provide algorithms specifically designed for the predictive kNN query.

2.3. Problem setting and optimization goals

We assume that the moving objects are stored on disk and indexed by a TPR-tree [19] (actually the variant TPR*-tree [24]). While there have been studies which assume all the objects are held in memory, we do not follow that setting because of the following reasons. First, there may not be enough memory to hold all the objects when the dataset is very large. Second, moving object monitoring is a continuously running process and system crashes may happen. Crash recoveries in a main-memory database incur significant overhead and are particularly troublesome [12]. Therefore, we assume that the dataset is stored on disk. Nonetheless, our techniques based on the new Transformed Minkowski concept will still apply even if the data are hold in a main-memory structure. We further assume a TPR-tree maintained because it is a generic and efficient disk-based indexing structure for moving objects. It can be used for various query types and it has been adopted in many previous studies as a basic structure.

Our goal is to minimize the query processing time, which consists of both disk I/O time and CPU time. Each disk page corresponds to a node in the TPR-tree and minimizing the number of node accesses is important. At the same time, CPU cost is also significant due to the following reasons. Moving object representation involves time parameters which add to the computational cost. Moreover, checking whether a moving object satisfies a spatial query predicate is more expensive than traditional spatial operations due to the movement of objects. Therefore, we try to reduce both I/O and CPU costs in our design and measure both of them in the experimental study.

If the moving objects send queries to a server for processing, the communication cost may also be an important issue, which has been addressed in some existing work such as [10]. However, this article focuses on the cost of query processing.

3. Preliminaries and related work

3.1. Representation of moving objects

Traditionally, a moving point's movement is represented by sampled positions on its trajectory. This approach requires frequent position updates, which impose heavy workload on the system. Sistla et al. [22] propose to model a moving point as a linear function of time t: $P(t)=P(t_{ref})+V(t-t_{ref})$, where $P(t_{ref})$ is the point's position at a reference time t_{ref}, and V is the point's velocity. This representation allows prediction of the point's future positions. When an object changes its speed, then it reports this change to the management system and the management system will update the object's position and speed information. Since most objects move in a linear fashion for short periods (e.g., cars moving on a road, people walking in the corridor of a building), this approach requires much fewer updates than the sampling-based approach and hence it is widely adopted by subsequent studies (e.g. [14,1,19,24,16,13]). For example, Kollios et al. [14] considered the problem of indexing moving points on 1-dimensional trajectories for predictive window queries. Agarwal et al. [1] addressed the problem in 2 and higher dimensions and proposed algorithms with good asymptotical performance. These results are mainly theoretical. Other studies aim at structures that yield good performance in practice. The TPR-tree [19] extends the R*-tree [2] by attaching time parameters to node regions so that the nodes can bound moving objects. The TPR*-tree [24] enhances the TPR-tree through a set of improved construction algorithms. As our proposed structure is based on the TPR/TPR*-tree, we describe them in more detail in Section 3.2 and 3.3, respectively. Indexing techniques based on non-R-tree structures are discussed in Section 3.4. There is no existing work on processing predictive range queries. Previous work on kNN queries is discussed in Section 3.6.

The above discussion has focused on moving points. For a moving object O of non-zero extent and irregular shape, O is represented by its *minimum bounding rectangle (MBR)* O_R at reference time t_{ref}, and its *velocity bounding rectangle (VBR)* O_V. An MBR O_R is of the form $\{O_{R\,1-}, O_{R\,1+}, O_{R\,2-}, O_{R\,2+}\}$[2] and a VBR O_V is of the form $\{O_{V\,1-}, O_{V\,1+}, O_{V\,2-}, O_{V\,2+}\}$. The VBR describes how each side of the MBR moves. Then the positions of the four sides of O can be calculated as linear functions of time. We call this representation of O a *time-parameterized bounding rectangle* (*TPBR*). In this article, moving objects assume this general representation through TPBRs and are not limited to moving points. The TPR-tree/TPR*-tree can handle

[2] As most real-life moving objects are in 2-dimensional spaces (such as cars or mobile phone users moving in a city), we focus on analyses in a 2-dimensional space in this article, although the results can be extended to higher-dimensional spaces. Subscripts "−" and "+" stand for lower bound and upper bound, respectively.

TPBRs straightforwardly, while the non-R-tree structures only apply to moving points. This is one of the reasons why we have chosen to use the TPR-tree as the indexing structure. Another reason is that the TPR-tree provides much more efficient range and kNN search performance than the non-R-tree structures as shown by our analysis in Section 3.4 and experimental results in Section 8.

3.2. The TPR-tree

Saltenis et al. [19] propose the TPR-tree (time parameterized R-tree), which is a variant of the R*-tree for indexing moving objects. Each node of the TPR-tree is a TPBR. A non-leaf node contains pointers to its child nodes. The MBR (VBR) of a non-leaf node bounds the MBRs (VBRs) of all its child nodes. A leaf node contains actual data objects and the leaf node's MBR (VBR) bounds these data objects' MBRs (VBRs). The TPR-tree essentially extends the traditional R*-tree by adding the VBR. We also use O to denote a TPBR or a TPR-tree node as they both represent moving objects.

Fig. 3 shows an example of a non-leaf node O of a TPR-tree at timestamp 0 ($O_R(0)$) and its predicted status at timestamp 2 ($O_R(2)$). Node O contains two child nodes a and b. The VBRs of a and b are {2,2,1,1} and {3,3,−1,1}, respectively, so the VBR of O is {2,3,−1,1}. The MBR of O at timestamp 0 ($O_R(0)$) is {1,3,3,6}. We can predict its MBR at time 2 by $O_R(0)+2\cdot O_V$, which results in $O_R(2)$, {5,9,1,8}. We also notice that the MBR may become larger than necessary as time goes on (see $O_R(2)$). In the TPR-tree, the MBR of a non-leaf node is tightened when there is an insertion or a deletion to the node because this would not add any extra node access. The insert and delete operations of the TPR-tree are similar to those of the R*-tree, with consideration of the effect of time.

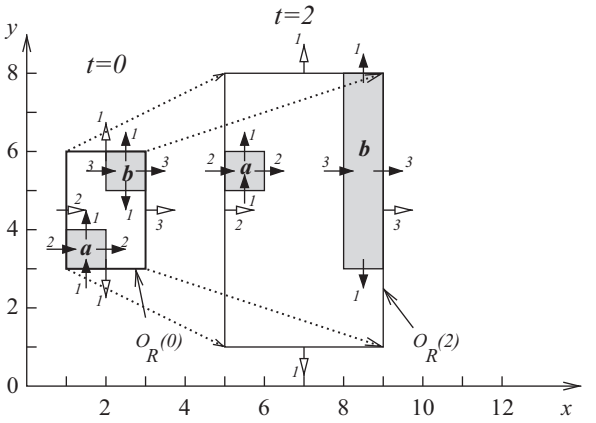

Fig. 3. Moving objects.

3.3. The TPR*-tree

Tao et al. [24] proposed the TPR*-tree that uses a set of improved algorithms to build the TPR-tree. They proposed a cost model for the predictive window query and their reported experiments show that the TPR*-tree achieves almost optimal performance according to the cost model. We briefly explain the cost model below (c.f. Fig. 4). Consider a moving object O and a moving window query WQ for the time interval [0,1]. The *sweeping regions* of O and WQ during the time interval are shown in Fig. 4(b) (the gray regions). To determine wether object O intersects WQ, Tao et al. first obtain the *transformed rectangle* O' with respect to WQ as follows (c.f. Fig. 4(b)). The MBR of O' on the i^{th} dimension is $\{O_{Ri-}-|WQ_{Ri}|/2, O_{Ri+}+|WQ_{Ri}|/2\}$; the VBR of O' on the i^{th} dimension is $\{O_{Vi-}-WQ_{Vi+},\ O_{Vi+}-WQ_{Vi-}\}$. Object O intersecting WQ during time interval [0,1] is equivalent to the sweeping region of the transformed rectangle of O' intersecting the center of WQ at timestamp 0 (which is a point). Therefore, determining whether O intersects WQ is easy and the probability of object O being accessed by the query WQ can be estimated through the area of the transformed rectangle. Based on this, they can estimate the cost of a query.

3.4. Indexes based on non-R-tree structures

Stripes: Stripes [16] use the dual transform to transform the positional representation of an object in a

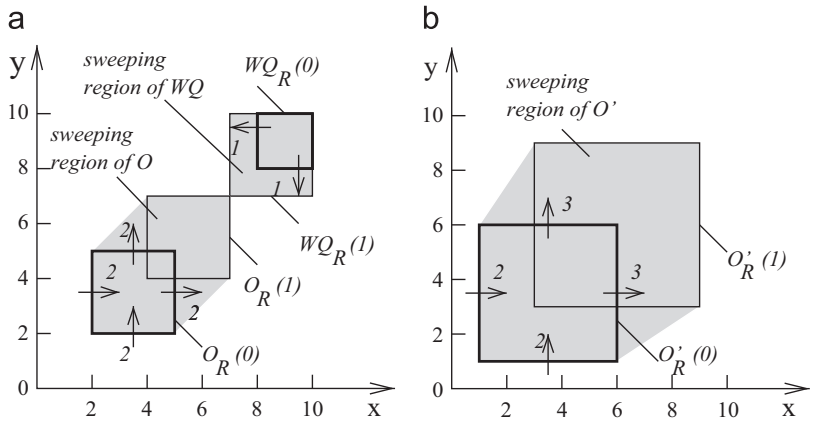

Fig. 4. Sweeping region of moving rectangle: (a) moving objects O, Q and (b) transformed rectangle O.

d-dimensional space to a $2d$-dimensional point. Recall that an object is represented by its position $P(t_{ref})$ at a reference time t_{ref}, and its velocity V (Section 2.3). Both $P(t_{ref})$ and V are d-dimensional vectors. In Stripes, these two d-dimensional vectors are combined to one $2d$-dimensional vector, $(V,P(t_{ref}))$. Then the moving objects represented by $2d$-dimensional vectors can be indexed by a traditional multidimensional indexing structure. In particular, Stripes use a disk-based quadtree [20]. Accordingly, queries are also transformed to the dual space. For example, a timeslice window query is transformed to a parallelogram and a moving window query is transformed to a union of two parallelograms. The search algorithms simply search the quadtree to find vectors in the transformed regions of the queries. Experiments reported in [16] show that Stripes have better update and query performance than the TPR*-tree. It is unclear how stripes can process kNN queries.

B^x-tree: The B^x-tree [13] is a structure proposed to handle frequent updates in moving object management while still having efficient query processing. The positions of moving objects in space are linearized according to their corresponding space-filling curve values and then organized in a B^+-tree. To support time parameters but avoid updating the whole tree at each timestamp, the time axis is partitioned into equal intervals called *phases* as shown in Fig. 5(a). Each phase is of length $\Delta t_{mu}/j$, where Δt_{mu} is the time duration between two *mandatory global updates*. An object updated at timestamp t_u is assigned a label timestamp $t_{lab} = \lceil t_u + \Delta t_{mu}/j \rceil_l$, where $\lceil x \rceil_l$ returns the nearest future label timestamp of x. By this means the objects are assigned to different partitions of the time axis. Consequently, all the points actually inserted at time 0 will be labelled timestamp $\Delta t_{mu}/2$ (j=2) and inserted to the first partition of the B^x-tree (the shaded partition in Fig. 5(a)); objects actually inserted during time $(0,\Delta t_{mu}/2]$ are labelled timestamp Δt_{mu} and inserted to the second partition of the B^x-tree, and so on. The predictive window query is processed through the *window enlargement* technique. Fig. 5(b) shows an example of how the window enlargement works. Suppose the current time is 0 and we issue a predictive timeslice window query WQ at time 2 (the solid rectangle). Consider moving points a and b (the black dots) stored in the B^x-tree, which are labelled timestamp 5. From their velocities as shown in the figure, we can infer their positions at timestamp 2, which are a^* and b^* (the circles). The window enlargement technique enlarges the query window WQ using the reverse velocities of a and b to get the query window at timestamp 5 (the dashed rectangle). In practice, histograms on a grid base are maintained for the maximum/minimum velocity of different portion of the data space and the query window is enlarged according to the maximum/minimum velocity in the region it covers. Therefore, a drawback of the B^x-tree is that, if only a few objects have very high speed, they would make the enlarged query window unnecessarily large for most of the points, while in the TPR-tree like structure, the high speed points have only local effects. The B^x-tree can also process the time-slice kNN query (but not the moving kNN query). The algorithm first issues a window query based on an estimated kNN distance, and then the window query is enlarged gradually until the kNN are found. This kNN algorithm suffers from the same aforementioned problem of overly enlarged query window. Moreover, the accuracy of kNN distance estimation gets worse for moving objects, which further deteriorates the performance.

B^{dual}-tree: Similar to the B^x-tree, the B^{dual}-tree [29] also exploits the space-filling curve. But instead of indexing just locations of moving objects, the B^{dual}-tree indexes the dual transform of both locations and velocities of moving objects. Yiu et al. [29] proposed a number of optimizations on the B^{dual}-tree. As a result, it has good performance in both updates and query processing.

ST^2B-tree: The ST^2B-tree [7] uses a normal B^+-tree to index moving objects with a key consisting of a time component and a space component. The time component is based on partitioning the time axis into equi-length buckets. The space component is based on a Voronoi-diagram partitioning of the space. The ST^2B-tree is designed with mechanisms for self-tuning so that it can adapt to changes of data density and distribution.

3.5. Analysis and benchmark

Most of the indexing techniques proposed can be classified into two categories depending on whether the

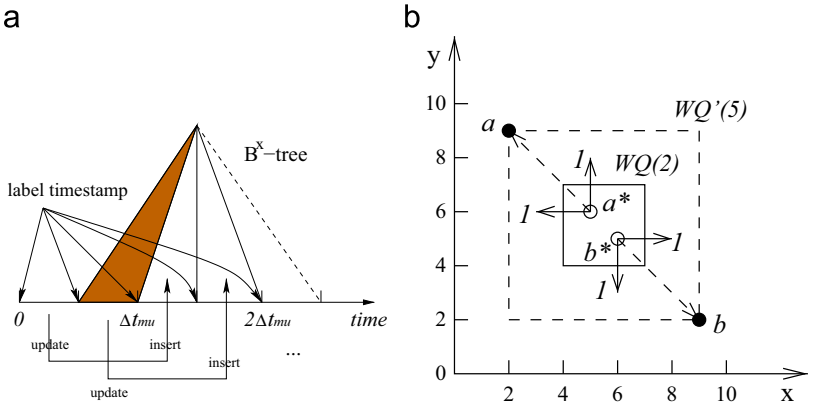

Fig. 5. The B^x-tree: (B^x-tree insertion and (b) B^x-tree window query.

indexing keys are in the primal or dual space. In the above discussed techniques, for example, Stripes and B^{dual}-tree are dual methods while the others are primal methods. Tao and Xiao [26] studied which approach is better from a theoretical point of view. Their analysis shows that different techniques may perform the best in different workloads. A primal index generally has low query cost but the performance may deteriorate over time, while a dual index has a cost much higher than the optimal value, but its performance will always remain reasonable disregard data property changes.

Chen et al. [6] performed a comparison on moving object indexing techniques in a practical setting. They developed a benchmark to generate datasets for testing indexing techniques on moving objects, and performed experiments to compare many notable existing techniques including the TPR-tree, the TPR*-tree, Stripes, the B^x-tree, the B^{dual}-tree as well as the R-tree, but with support for frequent updates [21]. The general conclusion is that different techniques may be the best for different metrics. Among all the techniques, the B^x-tree has the highest throughput and shortest update time. TPR-tree and TPR*-tree have very similar performance (except for concurrency control) and they have the shortest query processing time.

3.6. Algorithms for KNN queries on moving objects

There are many studies on processing continuous kNN queries on moving objects such as [3,4,17,11,28,10]. Since continuous kNN algorithms may be used for processing predictive kNN queries, we discuss them in more detail below. For simplicity, we will use NN (instead of kNN) to explain these algorithms.

Benetis et al. [3] use a TPR-tree to index the moving objects and then use a depth-first traversal algorithm to search the TPR-tree. The algorithm starts from the root of the tree. If a retrieved node is a non-leaf node, then the entries in the node are sorted according to the metric $M(R,q) = \int_{t_\vdash}^{t_\dashv} d_q(R,t)dt$, where $d_q(R,t)$ returns the squared distance between the query point Q and the point on a TPBR R that is the closest to Q at time t. Then the nodes are retrieved in the sorted order recursively down the tree. Here, sorting is a heuristic intended to retrieve the actual NN as early as possible. A pruning squared distance $dmin_q(t)$ is maintained for every timestamp in $[t_\vdash,t_\dashv]$ and initialized to infinity. During the traversal of the tree, if $d_q(R,t)$ of a node is greater than $dmin_q(t)$, then none of the children of this node needs to be retrieved; otherwise, the child nodes are retrieved in the sorted order of $M(R,q)$. For a leaf node, the algorithm determines for all timestamps in $[t_\vdash,t_\dashv]$, whether each entry has smaller $d_q(R,t)$ than $dmin_q(t)$. If yes, then this entry replaces the current NN candidate for the corresponding time interval; otherwise the entry is not an NN and is skipped. This algorithm returns the NN of every timestamp in $[t_\vdash,t_\dashv]$, so it actually returns the continuous NN answer (recall the discussion in Section 2.2). The algorithm involves solving quadratic inequalities for a list of time intervals, which is unnecessary for a predictive NN query. Later in the journal version [4], Benetis et al. also propose a best-first traversal algorithm and use the metric $min_{t\in[t_\vdash,t_\dashv]}d_q(R,t)$ to sort entries. The reported experiments show that the best-first search using the $min_{t\in[t_\vdash,t_\dashv]}d_q(R,t)$ metric performs the best among the variants, however, the gain is marginal.

Raptopoulou et al. [17] take the same paradigm as Benetis et al.'s work [3,4], that is, first returning continuous NN for the querying period $[t_\vdash,t_\dashv]$, and then updating answers for subsequent timestamps. Raptopoulou et al.'s work differ in how to find the answers. They first find the NN for the timestamp $t_\vdash$. Then the continuous NN for $[t_\vdash,t_\dashv]$ is derived by computing intersections between the initial NN and possible candidate objects for subsequent timestamps. Again, the intersection computation between the initial NN and many possible candidates is too expensive and unnecessary for the predictive NN query.

Iwerks et al. [11] use the event driven model to process the continuous NN query. Two kinds of events are considered: the *within event* (w-event) and the *order change event* (oc-event). The w-event affects the answer for the window query and the oc-event affects the answer for the NN query. Iwerks et al. find that window queries are easier to maintain than NN queries. Their method is to first find initial NN for $t_\vdash$, then to maintain the NN through processing oc-events in a maintained window query. Finding the initial NN involves computing w-events from all objects in the database for a window query and then compute the NN based on the objects in the window query. This is too expensive since all objects are retrieved.

The SEA-CNN [28] is an algorithm for processing multiple continuous NN queries on moving objects. It improves efficiency and scalability through incremental evaluation and shared execution. The moving objects and queries are organized using a grid structure and maintained in two tables separately. Multiple NN queries are executed by scanning the object and query tables. Then the retrieved objects and queries are joined to obtain results for all queries. The SEA-CNN has focused on how to continuously maintain the NN results and how to share computation, but not on how to obtain the initial NN or how to answer the predictive NN query.

Hu et al. [10] proposed a new framework for monitoring continuous spatial queries, including kNN queries, over moving objects. They consider the problem in a client-server environment and target minimizing the communication between the mobile clients and the server, which answers the queries. They use the concept of *safe region*, which is the region that keeps the answer of a query unchanged. The update and query reevaluation cost is greatly reduced by safe regions. The kNN algorithm is a best-first traversal algorithm adapted to leaf entries that may be either a point or a safe region. This algorithm cannot be applied to our problems since a different framework is assumed.

Yu et al. [30] and Mouratidis et al. [15] consider the continuous kNN query when the whole dataset can be held in main memory. Both papers assume objects update their positions periodically and are organized using linked lists based on a grid partition of the space. These

algorithms cannot be applied to our problems since we assume that the moving objects reside on disk.

Tao et al. [23] introduced the predictive kNN query (they call it the spatio-temporal kNN query) and gave an analysis on the expected nearest distance for the predictive kNN query, but they did not provide any algorithm to process the query. Tao et al. [25] introduced another type of query, called the time-parameterized (TP) queries, including the TPkNN query. The TPkNN query returns the current kNN set and its validity time range. In effect, it returns the first $\langle A_i, T_i \rangle$ pair (see the example of Fig. 2) of the continuous kNN query. Therefore, the TPkNN query is different from the predictive kNN query.

Recently, Zhang et al. [31] proposed a method to significantly reduce the time range for the processing of continuous queries. The proposed *time constraint processing* method applies to general continuous queries but is not helpful to predictive range or kNN queries.

To summarize, most studies focused on continuous kNN queries and their algorithms have focused on the continuous part, while the initial kNN set is obtained by a timeslice kNN query using traditional algorithms. Only Benetis et al.'s work [4] has proposed specific methods trying to improve the performance of obtaining the kNN set during a time interval and this work is also the most recent one. Therefore, we will compare our algorithm with Benetis et al.'s algorithm and the B^x-tree based kNN algorithm for predictive kNN queries in our experimental study.

4. The Transformed Minkowski Sum

In this section, we propose a method to determine whether a TPBR, that is, a moving rectangle intersects a moving circle, which is essential to processing range and kNN queries. Our method is based on a new concept called the *Transformed Minkowski Sum*. The Transformed Minkowski Sum is obtained by performing: (i) a coordinate transform according to the movement of the query, and then (ii) the Minkowski enlargement in the transformed coordinate system. This method can be further generalized to determine the intersection of two moving objects of any shapes.

As discussed in Section 3.3, a TPBR O intersecting a moving window query WQ is equivalent to (that is, necessary and sufficient condition of) the transformed rectangle O' intersecting the center of WQ (which is a point). This equivalent condition makes determining whether O intersects WQ much easier. One only needs to check whether O' contains the center of WQ. However, this approach cannot be straightforwardly extended to range or kNN queries, because now the query region is a circle instead of a rectangle. It is unclear how to enlarge a moving rectangular node by a *moving* circle. Our strategy to overcome this difficulty is to make the moving circle static first. Specifically, we apply a coordinate system transformation to make the query point static and located at the origin (the radius is still changing with time). The following example illustrates this process. Fig. 6 shows a TPR-tree node O, which is a TPBR, and a moving range query Q, which is a moving circle (with changing radius). Initially at timestamp 0, the MBR of the node $O_R(0)$ is {2,4,3,5} and the VBR of the node O_V is $\{-1,0,-1,1\}$; the query point Q is {9,9} and the velocity of the query point Q_V is $\{-3,-3\}$; the radius of the query r is 1 and the speed of the changing radius r_v is 1. The querying period Q_T is [0,1]. The initial states of both O and Q are drawn in thick solid lines. Their final states (at timestamp 1) are drawn in thin solid lines. At timestamp 1, O_R becomes {1,4,2,6}, the query point moves to {6,6} and its radius becomes 2. The gray regions represent the sweeping regions of the TPBR and the moving circle, respectively, during the period [0,1]. The 2-dimensional position space and the 2-dimensional velocity space compose a 4-dimensional space. At timestamp 0, the query point's coordinates in this 4-dimensional space is {9,9,−3,−3}. Based on these coordinates, we apply the coordinate transformation $\{-9,-9,+3,+3\}$ to the 4-dimensional space so that the query point is located at the origin and becomes static. As a result of the coordinate transformation, O_R becomes $\{-7,-5,-6,-4\}$ at timestamp 0 and the O_V becomes {2,3,2,4}; then we can obtain O_R at timestamp 1 in the transformed coordinate system, $\{-5, -2, -4, 0\}$. The TPBR and the moving circle in the transformed coordinate system are shown in Fig. 7.

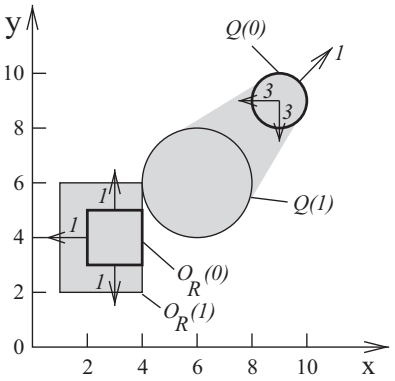

Fig. 6. A TPBR and a query circle.

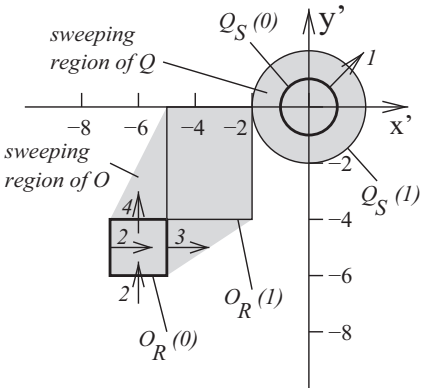

Fig. 7. Coordinate transformation.

After the coordinate transformation, we use the *Minkowski sum* [8] to derive the intersection between

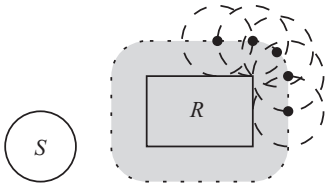

Fig. 8. The Minkowski sum.

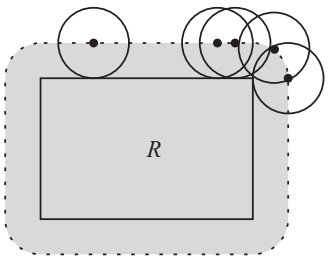

Fig. 9. Another way of obtaining the Minkowski sum.

the TPBR and the moving circle. The Minkowski sum of two sets A and B in a vector space is the union of adding every element of A to every element of B, that is, the set $A \oplus B = \{a+b : a \in A, b \in B\}$. For example, in Fig. 8, the Minkowski sum of the rectangle R and the circle S is the gray region (assuming the center of S is at the origin). Another way to obtain the same Minkowski sum is to roll S along the surface of R, and then the trajectory of the center of S is the boundary of the Minkowski sum of R and S as shown in Fig. 9. Here we actually enlarge R by the radius of S, therefore the Minkowski sum is also called the *Minkowski enlargement*. Observe Fig. 9. It is easy to prove that:

Lemma 1. *If R intersects any point of S, then the Minkowski enlargement (sum) of R with regard to S intersects the center of S, and vice versa.*

This lemma has been used before such as in [5]. We can further have the following lemma.

Lemma 2. *The closest distance between R and S, is the closest distance between the Minkowski enlargement (sum) of R with regard to S and the center of S.*

Proof. The proof is provided in Appendix A.

Continue with the example in Fig. 7. We can obtain the Minkowski sum of O and Q in the transformed coordinate system at every timestamp and union these Minkowski sums to get the sweeping region of the Minkowski sum. We call this Minkowski sum in the transformed coordinate system the *Transformed Minkowski Sum* of O with regard to Q (the formal definition is given later), denoted by $TMS(O,Q)$. Further, we denote the Transformed Minkowski Sum of O with regard to Q at timestamp t by $TMS(O,Q,t)$. The above process is illustrated in Fig. 10.

At any timestamp in Q_T, if $TMS(O,Q)$ intersects the origin (that is, the query point in the transformed coordinate system), then O intersects Q according to Lemma 1. In other words, during Q_T, if the sweeping region of $TMS(O,Q)$ intersects the origin, then O intersects Q. On the other hand, if the sweeping region of $TMS(O,Q)$ does not intersect the origin, then $TMS(O,Q)$ does not intersect the origin at any timestamp, and then O does not intersect Q during Q_T. Therefore, during Q_T, the sweeping region of $TMS(O,Q)$ intersecting the origin is equivalent to O intersecting Q. Consequently, we have the following theorem.

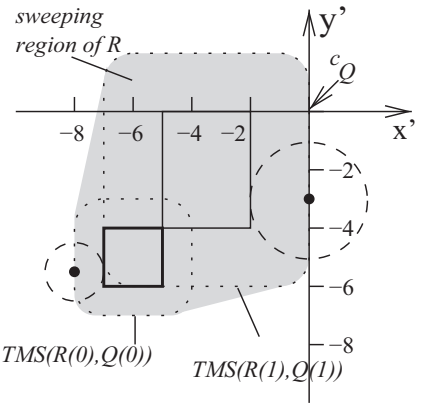

Fig. 10. The sweeping region of the Transformed Minkowski Sum.

Theorem 1 (*Equivalent condition*). *Given a TPBR O, a moving range query Q and a querying period Q_T, the sweeping region of $TMS(O,Q)$ intersecting the origin during Q_T is the necessary and sufficient condition of O intersecting Q during Q_T.*

The proof follows the early discussion.

Formally, the Transformed Minkowski Sum of a TPBR with regard to a moving circle is defined as follows:

Definition 5 (*Transformed Minkowski Sum*). Consider the 4-dimensional coordinate system G, which consists of 2 dimensions from the position space and two dimensions from the velocity space. Given a TPBR O and a moving range query Q, we apply the coordinate transformation $\{-Q_{C1}, -Q_{C2}, -Q_{V1}, -Q_{V2}\}$ to G and obtain the transformed coordinate system G'. The *Transformed Minkowski Sum* of O and Q is defined as the Minkowski sum of O with regard to Q in G'.

This definition can be generalized to the Transformed Minkowski Sum of two moving objects of any shapes. For two TPBRs, the Transformed Minkowski Sum becomes the transformed TPBR as discussed in Section 3.3.

In the rest of this article, we denote the sweeping region of TMS(O,Q) during time $[t_\vdash, t_\dashv]$ simply as $SR(O,Q,t_\vdash,t_\dashv)$, and it is formally defined as $SR(O,Q,t_\vdash,t_\dashv) = \cup_{t_\vdash \le t \le t_\dashv} TMS(O,Q,t)$.

5. Processing predictive range queries

In this section, we give our algorithm for processing predictive range queries based on the Transformed Minkowski Sum. There are three kinds of predictive range

```
RangeSearch(Node root, RangeQuery Q, Period Q_T)
1 A ← ∅ // the answer set
2 RangeSearchNode(root, Q, t⊢, t⊣) // [t⊢, t⊣] = Q_T
3 return A
End RangeSearch
```

Fig. 11. Algorithm *RangeSearch*.

```
RangeSearchNode(Node O, Q, t⊢, t⊣)
1 if O is a non-leaf node
2   for every entry e of O
3     if SR(e.TPBR, Q, t⊢, t⊣) intersects the origin
4       retrieve e.node
5       RangeSearchNode(e.node)
6 if O is a leaf node
7   for every entry e of O
8     if SR(e.TPBR, Q, t⊢, t⊣) intersects the origin
9       A ← A ∪ e.object
End RangeSearchNode
```

Fig. 12. Algorithm *RangeSearchNode*.

queries as discussed in Section 2.1. We only present the algorithm for the most general version, that is, the moving range query (Definition 3). Given a moving range query Q, we need to find all the objects intersecting Q in Q_T. According to Theorem 1, the equivalent condition of an object O intersecting Q during $Q_T=[t_{\vdash},t_{\dashv}]$ is $SR(O,Q,t_{\vdash},t_{\dashv})$ intersecting the origin. This equivalent condition enables us to use a tree traversal algorithm to process predictive range queries. The algorithm is called *RangeSearch* and runs as follows. It traverses the tree starting from the root. For every accessed node, the algorithm checks whether any child of the node intersects the query. If yes, the algorithm retrieves that child node and performs the above operations recursively until the leaf node level. Figs. 11 and 12 show the detailed steps of the algorithm.

This algorithm makes it possible to identify exactly those objects that intersect Q, which is not achieved by previous algorithms; issuing a predictive window query that bounds Q can only give a superset of the answer set and there is no existing way to tell which objects actually intersect Q.

The algorithm needs to determine if $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin, which is not a trivial derivation. We show this derivation in Section 5.2. Before that, we first prove that the algorithm is optimal for the predictive range query in terms of the number of node accesses in Section 5.1.

5.1. Optimality of algorithm RangeSearch

Lemma 3. *Any correct algorithm for the predictive range query must access every node whose TPBR intersects the query range during Q_T.*

Proof. For every node whose TPBR intersects the query range during Q_T, we must retrieve it to know whether it contains any object that satisfies the query. Otherwise, we may miss an object that should be in the answer set. □

Theorem 2. *The number of node accesses of algorithm RangeSearch is optimal for the predictive range query.*

Proof. RangeSearch only accesses a node O if O satisfies that $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin, which is equivalent to O's TPBR intersects the query range during Q_T. Therefore, RangeSearch only accesses the nodes that must be accessed according to Lemma 3. □

5.2. Determining whether $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin

We classify the motion of O_R into 4 basic cases by the directions that the opposite sides of O_R move in:

Case A: The left and the right sides move in opposite directions; the top and the bottom sides move in opposite directions, too.

Case B: The left and the right sides move in the same direction; the top and the bottom sides move in opposite directions. Without loss of generality, we assume both the left and the right sides move right.

Case B′: The left and the right sides move in opposite directions; the top and the bottom sides move in the same direction. Without loss of generality, we assume both the top and the bottom sides move upwards.

Case C: The left and the right sides move in the same direction; the top and the bottom sides move in the same direction, too. Without loss of generality, we assume both the left and the right sides move right, and both the top and the bottom sides move upwards.

When we consider the Minkowski sum of the rectangle with respect to a circle whose radius is changing with time, the motion of the rectangle becomes more complicated, because by comparing the speed of the radius with the velocities of the sides, the cases listed above are divided into more sub-cases. As a result, we have 10 sub-cases in total for the shape of $SR(O,Q,t_{\vdash},t_{\dashv})$ as shown in Figs. 13 and 14.[3] During Q_T, the initial (final) MBR of O is plotted as light (dark) gray regions in the figures, respectively. Each of the 10 sub-cases represents one particular position of the Minkowski sum of the initial MBR with respect to the initial radius and that of the final MBR with respect to the final radius. For each sub-case, we analyze the shape of $SR(O,Q,t_{\vdash},t_{\dashv})$ as follows.

Case A: (Fig. 13(a)). In this case, the final MBR encloses the initial MBR. As the radius does not decrease, the final Minkowski sum encloses the initial Minkowski sum and therefore $SR(O,Q,t_{\vdash},t_{\dashv})$ is the final Minkowski sum. We only need to determine if the final Minkowski sum intersects the origin.

Case B: There are two sub-cases as follow:

B.1 (Fig. 13(b)) In this case, the speed of the radius is less than the speed of the left side. $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by the initial Minkowski sum, the final

[3] Since $SR(O,Q,t_{\vdash},t_{\dashv})$ is obtained in the transformed coordinate system, the following discussions based on these figures refer to the transformed coordinate system.

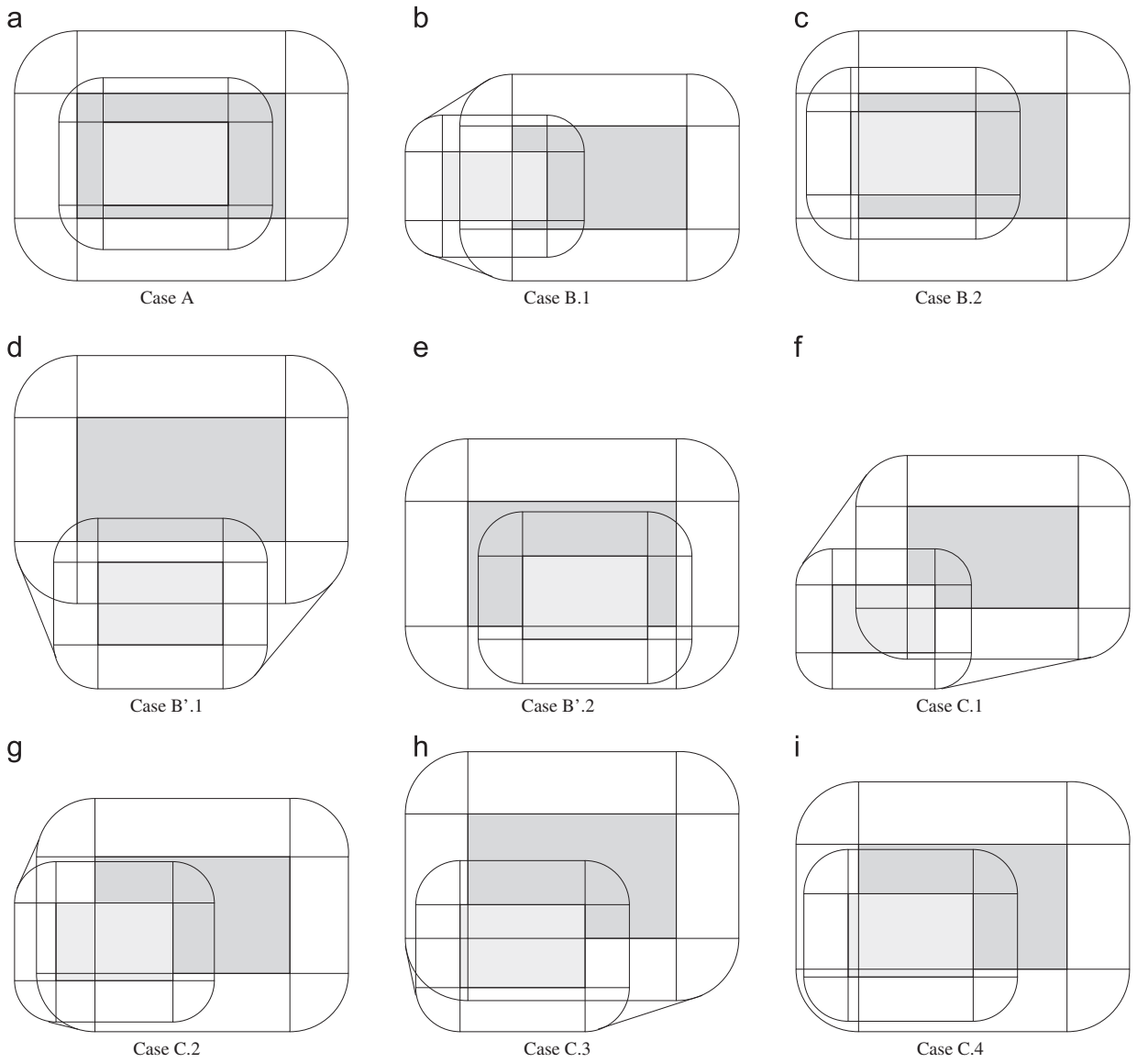

Fig. 13. Shape of $SR(O,Q,t_{\vdash},t_{\dashv})$ (Part1).

Minkowski sum, and two tangent line segments which are common to the pair of upper left arcs or the pair of lower left arcs. Thus to determine whether the origin intersects $SR(O,Q,t_{\vdash},t_{\dashv})$, we have to determine whether the origin intersects the initial Minkowski sum, the final Minkowski sum or the trapezoid formed by the two tangent line segments.

B.2 (Fig. 13(c)) In this case, the speed of the radius is greater than or equal to the speed of the left side. The final Minkowski sum encloses the initial Minkowski sum and therefore $SR(O,Q,t_{\vdash},t_{\dashv})$ is the final Minkowski sum.

Case B′: This case is very similar to the previous one, except for replacing the comparison between the speed of the radius and the speed of the left side by the comparison between the speed of the radius and the speed of the bottom side. There are also two sub-cases as follow. How to determine whether $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin is similar to case B, so we do not repeat the discussion here.

B′. (Fig. 13(d)) In this case, the speed of the radius is less 1 than the speed of the bottom side.

B′. (Fig. 13(e)) In this case, the speed of the radius is 2 greater than or equal to the speed of the bottom side.

Case C: In this case, we have to consider the comparison between the speed of the radius and the speed of the left side as well as the comparison between the speed of the radius and the speed of the bottom side. For the first three sub-cases described below, $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by

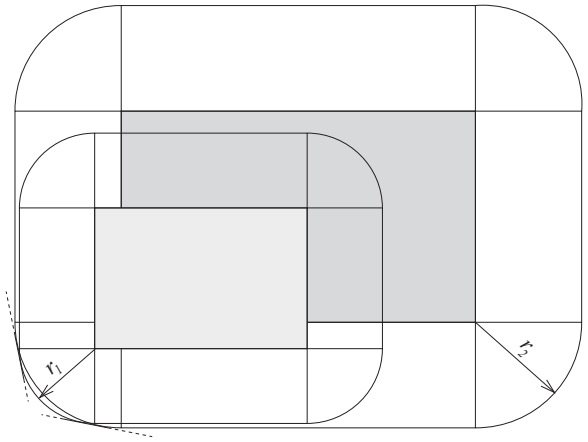

Fig. 14. Shape of $SR(O,Q,t_{\vdash},t_{\dashv})$ (Part 2), Case C.5.

the initial and the final Minkowski sum and two common tangent line segments. They differ in the position of the two tangent line segments.

C.1 (Fig. 13(f)) In this case, both the speed of the left side and the speed of the bottom side are greater than the speed of the radius. One tangent line segment is common to the pair of the upper left arcs and the other one is common to the pair of the lower right arcs.

C.2 (Fig. 13(g)) In this case, the speed of the left side is greater than the speed of the radius, while the speed of the bottom side is less than or equal to the speed of the radius. One tangent line segment is common to the pair of the upper left arcs; the other is common to the pair of the lower left arcs.

C.3 (Fig. 13(h)) In this case, the speed of the bottom side is greater than the speed of the radius, while the speed of the left side is less than or equal to the speed of the radius. One tangent line segment is common to the pair of the lower left arcs; the other is common to the pair of the lower right arcs.

C.4 (Fig. 13(i)) In this case, the root sum square of the speed of the left side and the speed of the bottom side is less than or equal to the speed of the radius. The final Minkowski sum encloses the initial Minkowski sum.

C.5 (Fig. 14) In this case, both the speed of the left side and the speed of the bottom side are less than or equal to that of the radius, but their root sum square is greater than the speed of the radius. This is a very special case, as the initial Minkowski sum has a small corner which is not covered by the final Minkowski sum; both tangent line segments are common to the lower left corner with different angles.

The shapes of $SR(O,Q,t_{\vdash},t_{\dashv})$ of all the above sub-cases fall into three classes: *Shape* 1, $SR(O,Q,t_{\vdash},t_{\dashv})$ is the final Minkowski sum, including cases A, B.2, B′. 2 and C.4; *Shape* 2, $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by the initial Minkowski sum, the final Minkowski sum and two common tangent line segments on two pairs of arcs, including cases B.1, B′. 1, C.1, C.2 and C.3; *Shape* 3, $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by the initial Minkowski sum, the final Minkowski sum and two common tangent line segments on the same pair of arcs, including only case C.5. Next, we show how to determine whether $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin for one case from each shape class. The other cases in the same shape class can be derived similarly. In what follows, let $\{O^{\dashv}_{R1-},O^{\dashv}_{R1+},O^{\dashv}_{R2-},O^{\dashv}_{R2+}\}$ be the final MBR and $\{O_{V1-},O_{V1+},O_{V2-},O_{V2+}\}$ the VBR of O; let r_1 (r_2) be the initial (final) radius of Q, respectively.

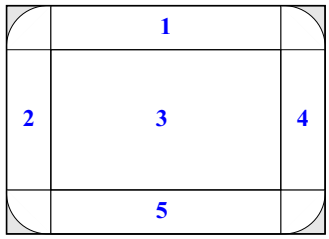

Fig. 15. $SR(O,Q,t_{\vdash},t_{\dashv})$ of Case A (Shape 1).

Shape 1: *Case* A, B.2, B′. 2 and C.4. We need to determine whether the origin is inside (that is, intersects) the final Minkowski sum. The origin is enclosed by the final Minkowski sum if and only if the following two conditions are both satisfied:

(1) $O^{\dashv}_{R1-}-r_2\le 0$ and $O^{\dashv}_{R1+}+r_2\ge 0$ and $O^{\dashv}_{R2-}-r_2\le 0$ and $O^{\dashv}_{R2+}+r_2\ge 0$;
(2) $O^{\dashv}_{R1-}\le 0\le O^{\dashv}_{R1+}$ or $O^{\dashv}_{R2-}\le 0\le O^{\dashv}_{R2+}$ or $min(|O^{\dashv}_{R1-}|,|O^{\dashv}_{R1+}|)^2+min(|O^{\dashv}_{R2-}|,|O^{\dashv}_{R2+}|)^2\le r_2^2$.

See Fig. 15. Condition (1) ensures that the origin lies within the rectangle in thick solid lines. Condition (2) ensures that the origin either lies in the rectangles 1–5, or if the origin falls in the four squares on the corners, it does not fall in the gray regions. Therefore, the two conditions together are equivalent to the origin being inside $SR(O,Q,t_{\vdash},t_{\dashv})$.

Shape 2: *Case* B.1, B′. 1, C.1, C.2 and C.3. As shown in Fig. 16, $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by the initial Minkowski sum, the final Minkowski sum, and four trapezoids (regions 6, 7, 10 and 11). We first use the method described in Case A to determine whether the origin is inside the initial Minkowski sum or the final Minkowski sum. Then we check whether the origin is inside the four trapezoids. Given an n-sided polygon in a 2-dimensional plane with vertices $(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)$, arranged in counter-clockwise order, the origin is inside this n-sided polygon if and only if (i) $x_iy_{i+1}-y_ix_{i+1}\ge 0,\forall i=1,\ldots,n-1$, and (ii) $x_ny_1-y_nx_1\ge 0$. Therefore, as long as we know the four vertices of a trapezoid, we can determine whether the origin is inside it by testing a few inequalities. For regions 6 and 7, we know their four vertices from the initial and final MBRs. For regions 10 and 11, we find their vertices as follows. Let w_1 (w_2) and h_1 (h_2) be the width and height of the initial (final) Minkowski sum, respectively, as shown in Fig. 16. We have: $r_1=r(t_{\vdash})=$

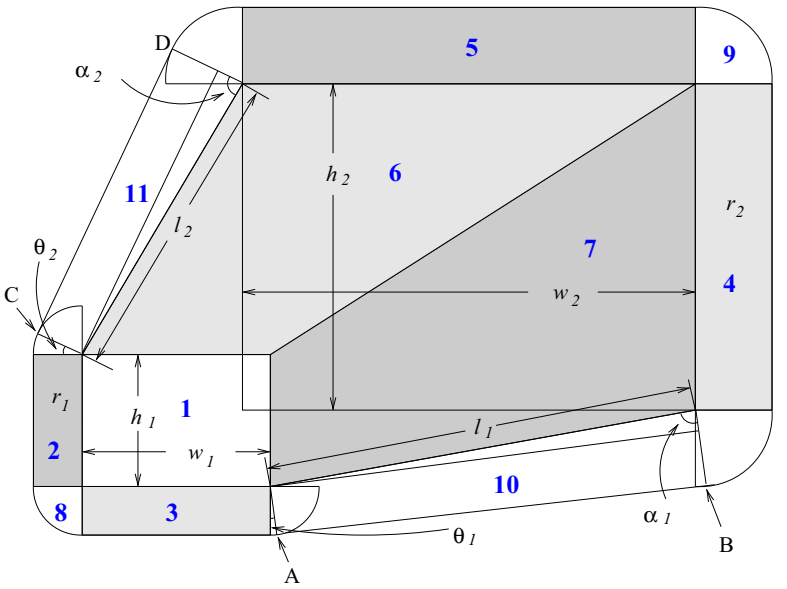

Fig. 16. $SR(O,Q,t_{\vdash},t_{\dashv})$ of case C.1 (Shape 2).

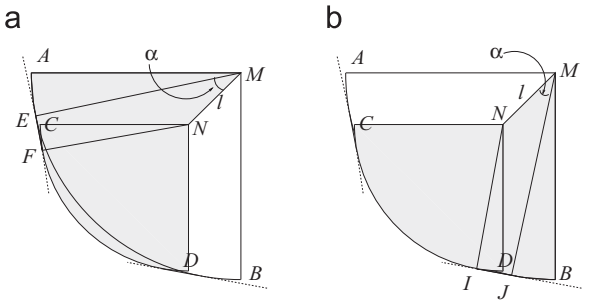

Fig. 17. Close look on the corner of Case C.5.

$r_{ref}+v_r(t_{\vdash}-t_{ref})$, $w_1=(O^{\dashv}_{R1+}-O^{\dashv}_{R1-})+(O_{V1+}-O_{V1-})(t_{\vdash}-t_{\dashv})$, $h_1=(O^{\dashv}_{R2+}-O^{\dashv}_{R2-})+(O_{V2+}-O_{V2-})(t_{\vdash}-t_{\dashv})$, $r_2=r(t_{\dashv})=r_{ref}+v_r(t_{\dashv}-t_{ref})$, $w_2=O^{\dashv}_{R1+}-O^{\dashv}_{R1-}$, and $h_2=O^{\dashv}_{R2+}-O^{\dashv}_{R2-}$.

Further, we define $l_1=\sqrt{|O_{V1+}|^2+|O_{V2-}|^2}(t_{\dashv}-t_{\vdash})$, and $l_2=\sqrt{|O_{V1-}|^2+|O_{V2+}|^2}(t_{\dashv}-t_{\vdash})$, where l_1 (l_2) is the length of one side of the trapezoid in region 10 (11), respectively, as shown in the figure. Then we can define the following angles: $\alpha_1=\cos^{-1}(r_2-r_1)/l_1$, $\alpha_2=\cos^{-1}(r_2-r_1)/l_2$, $\theta_1=\alpha_1-\tan^{-1}|O_{V2-}|/|O_{V1+}|$, and $\theta_2=\alpha_2-\tan^{-1}|O_{V2+}|/|O_{V1-}|$. With these angles known, we can easily find the four tangent points (A, B, C and D in the figure), two on each trapezoid . Two tangent points and two points from the initial and final MBRs compose the four vertices of each of the two trapezoids (regions 10 and 11).

If the origin is inside (that is, intersects) the initial or the final Minkowski sum, or any of the four trapezoids, then $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin, otherwise not.

Shape 3: *Case* C.5. Case C.5 is shown in Fig. 14 and a close view on its lower left corner is given in Fig. 17. $SR(O,Q,t_{\vdash},t_{\dashv})$ is bounded by the initial Minkowski sum, the final Minkowski sum, trapezoids MEFN (Fig. 17(a)) and MNIJ (Fig. 17(b)). We define $l=\sqrt{|O_{V1-}|^2+|O_{V2-}|^2}(t_{\dashv}-t_{\vdash})$ and $\alpha=\cos^{-1}(r_2-r_1)/l$, where l is the length of MN and α is the angle between the parallel sides of the trapezoids and MN. Then we have $\angle AME=\tan^{-1}|O_{V2-}|/|O_{V1-}|-\alpha$ and $\angle DMB=\tan^{-1}|O_{V1-}|/|O_{V2-}|-\alpha$. With these two angles known, we can compute the coordinates of points E, F, I and J, and once the four vertices of the trapezoids are known, we can test whether the origin is in them.

If the origin is inside (that is, intersects) the initial or the final Minkowski sum, or any of the two trapezoids, then $SR(O,Q,t_{\vdash},t_{\dashv})$ intersects the origin, otherwise not.

6. Processing predictive KNN queries

In this section, we give our algorithm for processing predictive kNN queries. There are three kinds of predictive kNN queries as discussed in Section 2.2. We only present the algorithm for the most general version, that is, the moving kNN query (Definition 4). Given a moving kNN query Q, we need to find the k objects that will be closest to the query point during the querying period Q_T. Moreover, we would also like to know when the k objects are closest to Q. For example, when a working robot needing maintenance finds out which maintenance robot will come closest to it within the next 10 min, the working robot also wants to know the exact time when maintenance robot is closest to it.

Observe that the only difference between the moving kNN query and the traditional kNN query is the function used to rank the objects, that is, $closest(O,Q,Q_T)$ rather than the Euclidian distance. Therefore, we can plug $closest(O,Q,Q_T)$ as the distance function into an existing kNN search algorithm based on R-trees. Two popular traditional kNN search algorithms are the depth-first

```
KNNSearchBF(Node root, KNNQuery Q, Period Q_T)
1  A ← ∅ // the answer set
2  pd ← ∞ // pruning distance
3  insert all entries of root to L /* L is a list sorted
         on closest(e.TPBR, Q, Q_T) */
4  do
5     remove the first entry e from L
6     if e.node is a non-leaf node
7        retrieve e.node
8        for every entry e' in e.node
9           if closest(e'.TPBR, Q, Q_T) ≤ pd
10             insert e' to L
11    if e.node is a leaf node
12       retrieve e.node
13       for every object O in e.node
14          < d_n, t_n > ← closest_time(O, Q, Q_T)
15          if |A| < k
16             A ← A ∪ < O, d_n, t_n >
17             if |A| = k
18                pd ← MaxClosest(A, Q).d_n
19          else if d_n < pd // already k candidates
20             A ← A − MaxClosest(A, Q)
21             A ← A ∪ < O, d_n, t_n >
22             pd ← MaxClosest(A, Q).d_n
23 while L is not empty and for the first entry e of L,
     closest(e.TPBR, Q, Q_T) ≤ pd
24 Return A
End KNNSearchBF
```

Fig. 18. Algorithm *KNNSearchBF*.

algorithm proposed by Roussopoulos et al. [18] and the best-first algorithm proposed by Hjaltason and Samet [9]. We only present the best-first algorithm, called *KNNSearchBF*, for predictive kNN queries as shown in Fig. 18. The depth-first algorithm (KNNSearchDF) can be obtained similarly by replacing the distance function in the traditional depth-first kNN search algorithm. Together with the k nearest objects, we also return the timestamps at which they are the kth nearest objects.

The input of KNNSearchBF are: the root node of the TPR*-tree *root*, the moving kNN query Q and the querying period Q_T. First, the answer set A is initialized to $\emptyset$ and the pruning distance pd is initialized to ∞. The algorithm traverses the tree nodes and data objects in order of their $closest(O,Q,Q_T)$, using a priority queue L to maintain the entries (pointers to nodes or objects) to be accessed. Initially the entries in the root node are all inserted in L and then entries in L are dequeued one after another. If a dequeued entry e points to a non-leaf node, the non-leaf node is retrieved and any entry e' in it is inserted back into L provided that $closest(e'.TPBR,Q,Q_T)$ is less than or equal to the pruning distance pd. If a dequeued entry e points to a leaf node, all objects in the leaf node are retrieved and the k objects with the smallest $closest(O,Q,Q_T)$ are maintained in a candidate set A. When the number of candidates reaches k, pd is accordingly updated as the largest $closest(O,Q,Q_T)$ of the objects in A. The function $MaxClosest(A,Q)$ returns the object in A that has the largest $closest(O,Q,Q_T)$. The above process continues until L becomes empty or $closest(e.TPBR,Q,Q_T) > pd$ for the next dequeued entry e. Then A is the answer set. In line 14, the function $closest\_time(O,Q,Q_T)$ returns a pair $\langle d_n, t_n \rangle$, where d_n is $closest(O,Q,Q_T)$ and t_n is the timestamp this closest distance happens.

We have yet to show: given O, Q and Q_T, how to calculate $closest(O,Q,Q_T)$ and the timestamp at which this distance happens. We derive $closest(O,Q,Q_T)$ in Section 6.2 and the timestamp in Section 6.3. Before those, we prove that algorithm KNNSearchBF is optimal in terms of the number of node accesses in Section 6.1.

6.1. Optimality of algorithm KNNSearchBF

We first introduce the concept of the kNN search circle, which is used in the proof of the optimality of the algorithm. In KNNSearchBF, node entries are accessed in increasing order of $closest(O,Q,Q_T)$, where an entry in a node is denoted by O and it corresponds to either a lower level node or a data object. Let d_k denote the $closest(O,Q,Q_T)$ of the kth NN for the predictive kNN query; d_k equals $MaxClosest(A,Q).d_n$ when KNNSearchBF terminates. The query point Q and d_k defines a moving circle (with Q as center and d_k as radius) which covers the searched space. We call this moving circle the *kNN search circle* (for a predictive kNN query).

Lemma 4. *Any correct algorithm for the predictive kNN query must access every node whose TPBR intersects the kNN search circle.*

Proof. For every node whose TPBR intersects the kNN search circle during Q_T, we must retrieve it to find out whether it contains any object that satisfies the query. Otherwise, we may miss an object that has smaller $closest(O,Q,Q_T)$ than d_k and hence should be in the answer set. □

Theorem 3. *The number of node accesses of algorithm KNNSearchBF is optimal for the predictive kNN query.*

Proof. Since KNNSearchBF accesses the node entries in increasing order of $closest(O,Q,Q_T)$ and when KNNSearchBF stops, the last accessed entry's $closest(O,Q,Q_T)$ is d_k, only the nodes with $closest(O,Q,Q_T)$ smaller than d_k are accessed. According to Lemma 4, these are the nodes that must be accessed to guarantee correctness. Therefore, KNNSearchBF only accesses the nodes that must be accessed. □

6.2. Closest distance

Theorem 4. *Given a TPBR O, a predictive kNN query Q and a querying period $Q_T = [t_\vdash, t_\dashv]$, the closest distance between Q and O during Q_T equals the closest distance between $SR(O,q,t_\vdash,t_\dashv)$ and the origin.*

Proof. Given Lemma 2 and the observation that coordinate transformation does not change distances between objects at any time point, then the closest distance

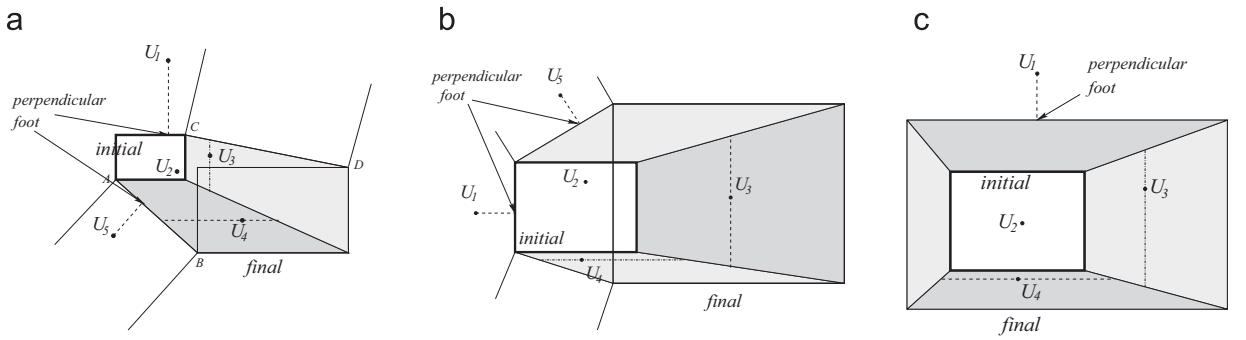

Fig. 19. The closest distance: (a) Case A; (b) Case B; and (c) Case C.

between Q and O at any timestamp t is the closest distance between $TMS(O,Q,t)$ and the origin. The closest distance between Q and O at all timestamps in Q_T is the minimum of the closest distances between $TMS(O,Q,t)$ and the origin for all timestamps $t \in Q_T$, which equals the closest distance between $\cup_{t_\vdash \le t \le t_\dashv} TMS(O,Q,t)$ and the origin, that is, the closest distance between $SR(O,Q,t_\vdash,t_\dashv)$ and the origin. □

Theorem 4 means that we just need to calculate the distance between $SR(O,q,t_\vdash,t_\dashv)$ and the origin to obtain $closest(O,Q,Q_T)$. The predictive kNN query is a moving point. Therefore, $SR(O,q,t_\vdash,t_\dashv)$ reduces to the sweeping region of the TPBR O and $TMS(O,Q,t_\vdash)$ ($TMS(O,Q,t_\dashv)$) becomes the initial (final) MBR of O, respectively, in the transformed coordinate system Fig. 19.[4] shows the typical cases with regard to the relative positions between the initial MBR and the final MBR of O. Any other case is similar to one of them, so we only present how to obtain the distance between $SR(O,q,t_\vdash,t_\dashv)$ and the origin for these cases. Fig. 19(a) is the case where each pair of parallel sides is moving in the same direction. Fig. 19(b) is the case where one pair of parallel sides is moving in the same direction, while the other pair is moving in opposite directions. Fig. 19(c) is the case where each pair of parallel sides is moving in opposite directions. In this case, $SR(O,q,t_\vdash,t_\dashv)$ is totally contained in the final MBR.

For cases A and B, $SR(O,Q,t_\vdash,t_\dashv)$ is a hexagon bounded by two rectangles and two line segments. For case C, $SR(O,Q,t_\vdash,t_\dashv)$ is a rectangle. For any case, we can easily find out all vertices of $SR(O,Q,t_\vdash,t_\dashv)$ and check whether the origin is inside $SR(O,Q,t_\vdash,t_\dashv)$ using the n-sided polygon method described in Section 5.2. If yes, then $closest(O,Q,Q_T)$ is 0; otherwise, the closest distance must be one of the following:

(1) The closest distance to the initial MBR,
(2) the closest distance to the final MBR, or
(3) the closest distance to one of the two line segments.

We calculate all these three distances and the minimum of them is the closest distance between $SR(O,Q,t_\vdash,t_\dashv)$ and the origin. For example, if the origin is at position U_1 in Fig. 19(a) or (b), the closest distance (from the origin) is to the initial MBR. If the origin is at position U_1 in Fig. 19(c), the closest distance is to the final MBR. If the origin is at position U_5 in Fig. 19(a) or (b), the closest distance is to a line segment.

The closest distance to an MBR is the smallest one among the closest distances to the four sides of the MBR. Every side of the MBR is a line segment. Therefore, all the above closest distances reduce to the calculation of the closest distance to a line segment.

For a line segment L with two ends (x_1,y_1) and (x_2,y_2), we draw a perpendicular line to L from the origin and let the perpendicular foot be Z. Z lies on the line segment if and only if the origin is bounded by the two normals of line L at (x_1,y_1) and (x_2,y_2), or mathematically, $((x_2-x_1)x_1+(y_2-y_1)y_1)((x_2-x_1)x_2+(y_2-y_1)y_2)<0$. In this case, the closest distance is $|x_1y_2-x_2y_1|/\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$. If Z is not on the line segment, the closest distance is the smaller one of the distances to the two ends of L, that is, $\min\left(\sqrt{x_1^2+y_1^2},\sqrt{x_2^2+y_2^2}\right)$.

For example, in Fig. 19(a), point U_5 is bounded by the two normals of the line segment AB, so the closest distance from U_5 to AB is the distance from U_5 to the perpendicular foot. Point U_1 is not bounded by the two normals of the line segment CD, so its closest distance to CD is the distance to the nearer end C. However, U_1C is larger than the distance of U_1 to the initial MBR, therefore the closest distance is calculated using the initial MBR.

6.3. The timestamp for the closest distance

In addition to the closest distance between Q and O, we are also interested in the timestamp when this closest distance happens. There are two cases:

(1) If O intersects Q during Q_T, we request the first timestamp at which the intersection happens.
(2) If O does not intersect Q, we request the timestamp at which $closest(O, Q, Q_T)$ happens.

We still consider the transformed coordinate system, so Q is at the origin. For case (1), if the initial MBR contains the origin (for example, the origin is at point U_2 in Fig. 19(a), (b) or (c)), then the requested timestamp is

[4] The following discussions based on this figure refer to the transformed coordinate system.

$t_{\vdash}$. Otherwise, the remaining part of $SR(O,Q,t_{\vdash},t_{\dashv})$ (the gray regions in Figs. 19(a), (b) and (c)) is divided into two to four trapezoids by the trails of the vertices of the MBR. The origin must be in one of these trapezoids. The requested time is the time when the moving MBR touches the origin, which is the time when one of the parallel sides of a trapezoid "sweeps" over it. Take point U_3 in Fig. 19(a), (b) or (c) for example. The parallel sides of the trapezoid that contains it are vertical. The requested timestamp is thus when the vertical side of the trapezoid meets U_3 (the vertical dashed line). For point U_4 in the figures, the requested timestamp is when the horizontal side of the trapezoid touches U_4. Therefore, to find out the requested timestamp, we first identify the trapezoid the origin is in and the corresponding parallel side. Then we calculate the requested timestamp according to the initial position and the velocity of the side.

For case (2), the closest distance (from the origin) must be to the initial MBR, the final MBR, or two line segments as discussed in the previous subsection. If the closest distance is to the initial MBR or the final MBR, the timestamp is $t_{\vdash}$ or $t_{\dashv}$, respectively. Point U_1 in Fig. 19(a), (b) or (c) is an example. Otherwise, the closest distance is to one of the two line segments on the boundary of $SR(O,Q,t_{\vdash},t_{\dashv})$. It happens when the moving MBR touches the perpendicular foot from the origin to its closest line segment on $SR(O,Q,t_{\vdash},t_{\dashv})$. Point U_5 in Fig. 19(a) or (b) is an example. As explained in Section 6.2, we can find the position of the perpendicular foot from U_5. Then the timestamp can be obtained according to the initial position of the MBR and the velocities of the sides of the MBR. The perpendicular foot from U_5 to line segment AB divides AB into two parts of lengths 1:2. Therefore, the timestamp when the closest distance happens is $(2t_{\vdash}+t_{\dashv})/3$. If the perpendicular foot is not on the line segment, the closest distance must be to either the initial or the final MBR.

7. Cost model

In this section, we develop a cost model to estimate the cost of predictive range queries, which again makes use of the Transformed Minkowski Sum and its sweeping region. We also discuss how the cost model for predictive range queries can help estimate the cost of predictive kNN queries.

7.1. Cost model for predictive range queries

To compute the expected number of nodes accessed by a predictive range query Q, we sum up the probability of every node intersecting Q. According to Theorem 1, a node O intersecting Q during the querying period $Q_T[t_{\vdash},t_{\dashv}]$ is equivalent to $SR(O,Q,t_{\vdash},t_{\dashv})$ intersecting the origin. Assume that the center of Q distributes uniformly in the data space. Then the probability of $SR(O,Q,t_{\vdash},t_{\dashv})$ intersecting the origin equals the area of $SR(O,Q,t_{\vdash},t_{\dashv})$ divided by the area of the data space. Without loss of generality, let the area of the data space be 1. Let $Area(g)$ denote the area of a region g. Then we have the following theorem.

Theorem 5 (*Cost model for the predictive range query:*). *Let Q be a predictive range query whose center is uniformly distributed in the data space, then the expected number of node accesses for answering Q is*

$$Cost(Q)=\sum_{every\ node\ O} Area(SR(O,Q,t_-,t_+)). \quad (1)$$

The proof is clear from the above discussion. Next we show how to compute $Area(SR(O,Q,t_-,t_+))$. We still consider the sub-cases categorized into three classes as described in Section 5.2 and use the same notation as those used there.

Shape 1: *Case* A, B.2, B′. 2 and C.4. $SR(O,Q,t_-,t_+)$ is the final Minkowski sum as shown in Fig. 15. $Area(SR(O,Q,t_-,t_+))$ is the sum of the area of the rectangle (dark gray part), the area of the four marginal rectangles (light gray part) and the four quarter sectors.

$$Area(SR(O,Q,t_-,t_+))=(O^{\dashv}_{R1+}-O^{\dashv}_{R1-})(O^{\dashv}_{R2+}-O^{\dashv}_{R2-})+2(O^{\dashv}_{R1+}-O^{\dashv}_{R1-})r_2+2(O^{\dashv}_{R2+}-O^{\dashv}_{R2-})r_2+\pi r_2^2.$$

Shape 2: *Case* B.1, B′. 1, C.1, C.2 and C.3. As shown in Fig. 16, $SR(O,Q,t_-,t_+)$ consists of 11 regions: the initial MBR (region 1), four marginal rectangles surrounding the initial and final rectangles (regions 2–5), two trapeziums (regions 6 and 7), two sectors (regions 8 and 9) and two irregular shapes (regions 10 and 11). The two irregular shapes here are the most complicated parts. Each of them is divided further into two sectors and one trapezium. The lower right irregular shape is divided into two sectors whose areas are $\frac{1}{2}\theta_1 r_1^2$ and $\frac{1}{2}(\pi/2-\theta_1)r_2^2$, and one trapezium whose area is $\frac{1}{2}(r_1+r_2)l_1\sin\alpha_1$. The upper left irregular shape is divided into two sectors whose areas are $\frac{1}{2}\theta_2 r_1^2$ and $\frac{1}{2}(\pi/2-\theta_2)r_2^2$, and one trapezium whose area is $\frac{1}{2}(r_1+r_2)l_2\sin\alpha_2$. Computation of the areas of the other regions are straightforward, so we provide the result as follows:

$$\begin{aligned}Area(SR(O,Q,t_-,t_+))&=w_1h_1+r_1h_1+r_1w_1+r_2h_2+r_2w_2\\&+\frac{1}{2}(w_1+w_2)O_{V2+}(t_+-t_-)\\&+\frac{1}{2}(h_1+h_2)O_{V1+}(t_+-t_-)\\&+\frac{1}{4}\pi r_1^2+\frac{1}{4}\pi r_2^2+\frac{1}{2}(\theta_1+\theta_2)r_1^2\\&+\frac{1}{2}(\pi-\theta_1-\theta_2)r_2^2+\frac{1}{2}(r_1+r_2)l_1\sin\alpha_1\\&+\frac{1}{2}(r_1+r_2)l_2\sin\alpha_2,\end{aligned}$$

where $r_1,r_2,w_1,h_1,w_2,h_2,l_1,l_2,\alpha_1,\alpha_2,\theta_1$ and θ_2 have been derived in Section 5.2.

Shape 3: *Case* C.5. Case C.5 is shown in Fig. 14. The total area of other regions than the lower left corner is

$$(O^{\dashv}_{R1+}-O^{\dashv}_{R1-})(O^{\dashv}_{R2+}-O^{\dashv}_{R2-})+2(O^{\dashv}_{R1+}-O^{\dashv}_{R1-})r_2+2(O^{\dashv}_{R2+}-O^{\dashv}_{R2-})r_2+\frac{3}{4}\pi r_2^2. \quad (2)$$

The lower left corner as shown in Fig. 17 can be calculated by adding up the two light gray regions (one in Fig. 17(a) and one in Fig. 17(b)), and then subtracting their intersection. Their intersection is the quarter sector of the smaller radius (among the initial radius and the final radius). Each light gray region is divided into two sectors and one trapezium. The area of the light gray region in Fig. 17(a) is $\frac{1}{2}\angle AME\cdot r_2^2+\frac{1}{2}\angle FND\cdot r_1^2+\frac{1}{2}(r_1+r_2)l\sin\alpha$; the area of the light gray region in Fig. 17(b) is

$\frac{1}{2}\angle BMJ \cdot r_2^2 + \frac{1}{2}\angle INC \cdot r_1^2 + \frac{1}{2}(r_1+r_2)l\sin\alpha$, where $l=\overline{MN}$, and $\alpha=\angle EMN=\angle JMN$. Note that $\angle AME+\angle BMJ=\pi/2-2\alpha$ and $\angle FND+\angle INC=\pi/2+2\alpha$, so the sum of the area of the two light gray regions is $\frac{1}{2}(\pi/2+2\alpha)r_1^2+\frac{1}{2}(\pi/2-2\alpha)r_2^2+(r_1+r_2)l\sin\alpha$. By subtracting the area of their intersection $\pi/4r_1^2$, the total area of the lower left corner is

$$\alpha r_1^2+\frac{1}{2}(\pi/2-2\alpha)r_2^2+(r_1+r_2)l\sin\alpha. \quad (3)$$

By adding (2) and (3), we get

$$Area(SR(O,Q,t_-,t_+)) = (O^{\dashv}_{R1+}-O^{\dashv}_{R1-}+2r_2)(O^{\dashv}_{R2+}-O^{\dashv}_{R2-}+2r_2) + (\pi-4)r_2^2+\alpha r_1^2-\alpha r_2^2+(r_1+r_2)l\sin\alpha,$$

where $l=\sqrt{|O_{V1-}|^2+|O_{V2-}|^2}(t_+-t_-)$, and $\alpha=\cos^{-1}(r_2-r_1)/l$.

7.2. Discussion on the cost model of predictive kNN queries

As explained in the proof of Theorem 3, for a predictive kNN query Q, only the nodes with $closest(O,Q,Q_T)$ smaller than d_k are accessed. Let t_k be the time when the kth NN is found. Consider a range query Q' that has Q as its center, Q_V as its velocity, $[t_c,t_k]$ as the querying period Q_T, d_k as the radius and r_v=0 (i.e., the radius does not change). We can prove the following theorem.

Theorem 6. *During Q_T, a moving object O intersects Q' if and only if O's $closest(O,Q,Q_T)$ is less than or equal to d_k.*

Proof. If O intersects Q' during Q_T, then there must exist a timestamp in Q_T when O has a distance to Q smaller than or equal to d_k, so O's $closest(O,Q,Q_T)$ is less than or equal to d_k. On the other hand, if O's $closest(O,Q,Q_T)$ is less than or equal to d_k, O must intersect Q' when $closest(O,Q,Q_T)$ happens. □

The above theorem implies that Q' accesses only and all the nodes with $closest(O,Q,Q_T)$ less than or equal to d_k, which are the same nodes accessed by Q using our KNNSearchBF algorithm. If we know d_k, then we can use Q' and the cost model for predictive range queries to estimate the number of node accesses for predictive kNN queries. However, estimating kNN distance has been a difficult problem even for static objects [5,27]. Little work has been done on estimating kNN distance for moving objects. A full investigation on this topic is beyond the scope of this paper and we defer it to future work.

8. Experiments

This section reports our experimental results. We present the results on our predictive range search algorithm in Section 8.1 and the results on our predictive kNN search algorithm in Section 8.2. Finally we evaluate our cost model in Section 8.3.

We have generated datasets and queries using the benchmark data generator proposed by Chen et al. [6]. The parameters of the datasets are described as follows. The moving objects are populated in a 2-dimensional 100,000 × 100,000 coordinate space and 100 "hotspots" are uniformly distributed in the space. The moving objects gather around these hotspots and their distances to the hotspots follow a Gaussian distribution. By default, we use a dataset with 100,000 moving objects. There are 10 ring-shaped speed zones around each hotspot and the highest speed is 100. The speeds of objects are generated according to the speed zone and also in a way that guarantees the Gaussian distribution (please refer to [6] for more details). The horizon H of the TPR*-tree is 120 time units. Note that the benchmark data generator does not insert the 100,000 objects all at once at timestamp 0. Instead, their initial insertions are spread in time [0, 120] and there are updates of already inserted objects interleaved with the insertions. When the 100,000 objects are all inserted at timestamp 120, about 80,000 updates besides the insertions have been performed.

The centers of the queries follow the same distribution as the data. For every set of the experiments, we run 100 queries generated as above and report the average I/O and CPU time as the results. We will describe the query workloads as we discuss each particular experiment. The tree node size is 4096 bytes. All the techniques were implemented in C++. All the experiments were run on a desktop computer with Intel Duel Core 2.4 GHz CPU and 2 GB RAM.

8.1. Evaluation of the range search algorithm

In this subsection, we evaluate the performance of our proposed range search algorithm RangeSearch presented in Section 5. Since there was no previous study on the range query, a naive method is to first perform a window query that is the minimum TPBR of the range query, and then check all the retrieved objects against the range query to remove false positives. However, it should be noted that the false positives have to be pruned using our proposed method in Section 5 because no existing method can determine whether a TPBR intersects a moving range query. To process the window query, we use two recent algorithms. One is the window query algorithm for the TPR*-tree [24], the other is the window query algorithm for the B^x-tree [13]. We compare these two approaches with our algorithm RangeSearch.

In this set of experiments, the locations of the centers of the range queries follow the data distribution. The centers' speeds are randomly distributed in [0, 100]. The query radii are randomly distributed in [0, Q_{Rmax}]. The default value for Q_{Rmax} is 5,000. The starting time t_s for the querying period is randomly generated in the range [120, 240]. We first let the length of the querying period Q_T be 0. The I/O and CPU costs as functions of the number of updates are plotted in Fig. 20. As there are about 80,000 updates at the time when all objects are inserted (i.e., 120), the horizontal axis starts with 80,000 updates. In the figure, "TW" represents the TPR*-tree window query based approach; "TR" represents our range search algorithm; "BW" represents the B^x-tree window query based approach. We observe that TR always has less I/Os than TW and BW. This is because we only access the pages that intersect the query range while TW and BW access all

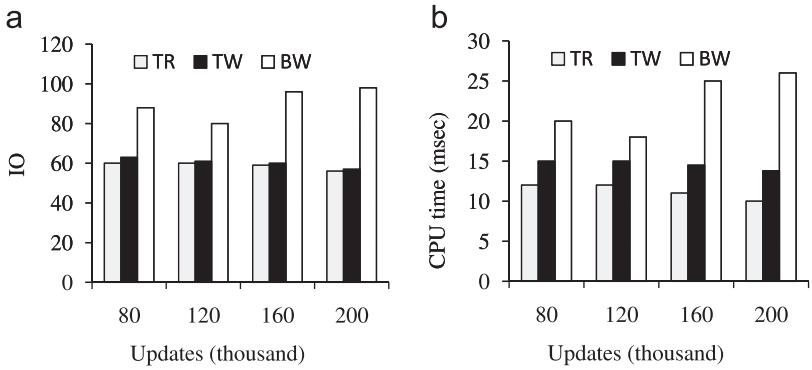

Fig. 20. Range queries, varying updates: (a) page accesses and (b) CPU time.

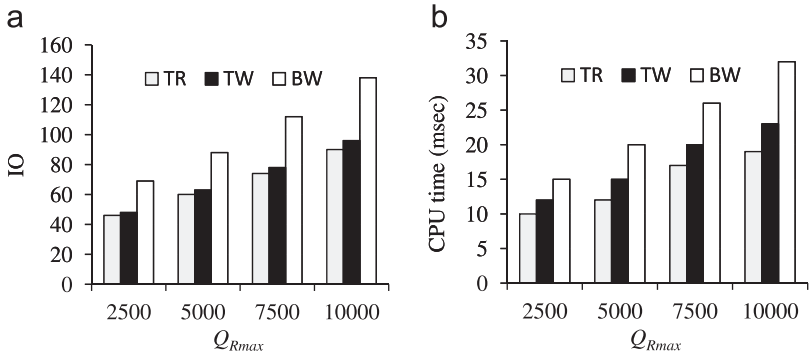

Fig. 21. Range queries, varying $Q_{R_{max}}$: (a) page accesses and (b) CPU time.

the pages that intersect the TPBR of the query range, which is always larger than the query range. BW has the largest number of I/Os because it needs to expand the query window according to the highest speed in the query area to guarantee no false dismissals; this pessimistic approach results in a actual search range larger than the necessary minimum query window. The CPU time plot has a similar trend to the I/Os in general. BW accesses more search space and therefore has more computation than TW and TR. We also observe that TW and TR have similar CPU time. Although the analysis in Section 5.2 for TW seems complicated, actually only a few inequalities need to be checked to decide whether a node needs to be accessed for each case. We can see that TW has the best overall performance among all the competitive techniques. Interestingly, both the IO and CPU time of TR and TW decrease slightly as updates increase. This may be because the insertions tighten the tree nodes and result in better performance. The costs of B^x-tree sometimes increase and sometimes decrease as the number of updates change (which corresponds to time change). This phenomenon conforms with the periodic behavior of B^x-tree's query performance observed in other studies [29,6].

We also compare the three techniques using other query workloads. We vary Q_{Rmax} from 2500 to 10,000 at the step of 2500 to see the effect of the query size. We use the dataset with 80,000 updates while the other parameters remain the same as above. The I/O and CPU costs as functions of Q_{Rmax} are shown in Fig. 21. It is expected that the costs of all the techniques increase as Q_{Rmax} increases. Their relative performance has the same trend as the previous experiment.

Next, we vary the length of the querying period Q_T from 0 to 60. Q_{Rmax} is 5,000 and the other parameters remain the same as above. The result is shown in Fig. 22. It is expected that the costs of all the techniques increase as Q_T increases. Again, their relative performance has the same trend as the previous experiment.

In summary, our RangeSearch algorithm based on the Transformed Minkowski Sum is the only one that can return the exact answer to predictive range queries, and moreover, it is more efficient than other techniques.

8.2. Evaluation of the KNN search algorithm

In this subsection, we evaluate the performance of our proposed kNN search algorithms KNNSearchBF and KNNSearchDF presented in Section 6. As discussed in Section 3, the most reasonable competitors are the B^x-tree based kNN algorithm (we call it *B-KNN*) and the continuous kNN algorithm proposed by Benetis et al. [4] (we call it *BJKS-KNN*). Therefore, we compare our algorithms with these two algorithms in the experiments. For BJKS-KNN, the best-first traversal with the $\min_{t\in[t_c,t_{\dashv}]} d_q(R,t)$ metric is used since it is reported to be the best variant in [4]. For the first two sets of

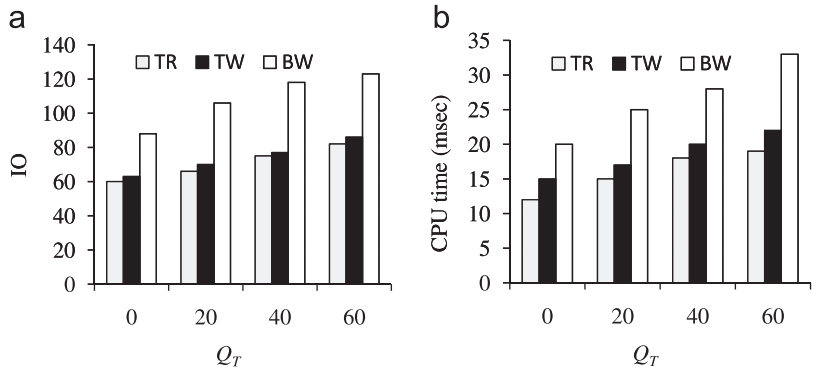

Fig. 22. Range queries, varying Q_T: (a) page accesses and (b) CPU time.

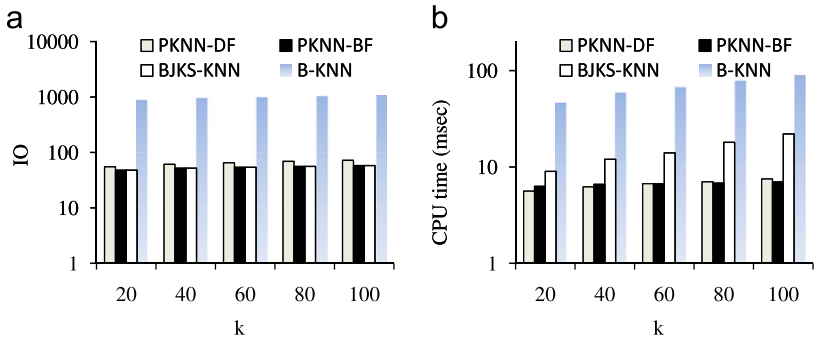

Fig. 23. KNN queries, varying k: (a) page accesses and (b) CPU time.

experiments, the length of Q_T is set to 0 so that the continuous nature of BJKS-KNN does not incur any unnecessary cost for predictive queries, but we will set Q_T as non-zero to see the effect in the remaining experiments. The locations of the kNN queries follow the data distribution. The queries' speeds are randomly distributed in [0, 100].

As discussed in Section 3.4, B-KNN can only process time-slice kNN queries rather than moving kNN queries. For the sake of performance comparison, we let all the kNN queries be static in this first set of experiments. The length of Q_T is 0 and the starting time t_s of Q_T is uniformly distributed in the period [120, 240]. The I/O and CPU costs as functions of k are plotted in Fig. 23. "PKNN-BF" and "PKNN-DF" represent our algorithms KNNSearchBF and KNNSearchDF, respectively. We observe that PKNN-BF has fewer I/Os than PKNN-DF because PKNN-BF is optimal in terms of node accesses. PKNN-BF has the same I/Os as BJKS-KNN. This is because when the length of Q_T is 0, the query becomes a snapshot query and the $\min_{t\in[t_{l^-},t_{u^+}]} d_q(R,t)$ metric also becomes I/O optimal for snapshot queries. All the above three algorithms have much fewer I/Os than B-KNN. The reason is as follows. The B^x-tree enlarges the query window according to the object of the highest speed, which may lead to unnecessarily large search range. Its kNN query performance may be even worse since it depends on an initial estimate of the kNN distance, which may be larger than the actual kNN distance. In terms of CPU cost, both PKNN-DF and PKNN-BF show notable improvements over BJKS-KNN. The reason is as follows. Our algorithms only need to compute *closest*(O,Q,Q_T) between entries and the query in the tree traversal, which can be obtained through computing a few distances between points and line segments (Section 6.2), while BJKS-KNN needs to solve a number of quadratic inequalities (Section 3.6). This was verified by a profiling analysis. PKNN-BF has slightly higher CPU cost than PKNN-DF when k is small and then lower CPU cost than PKNN-DF when k is large. The reason is that PKNN-BF needs to maintain a priority queue for the NN candidates. When k is small, the benefit of retrieving slightly fewer pages is less than the overhead of maintaining the queue, but the benefit overweighs the overhead when k is large. The B^x-tree is much worse than the other algorithms in terms of CPU time for similar reasons discussed above. We do not show the results of B-KNN in the remaining experiments for presentation purposes. The I/O and CPU cost of all techniques increase as k increases, because the larger the k, the more objects need to be accessed and evaluated.

In the second experiment, we see how the performance changes with updates. As in the predictive range query experiments, we vary the number of updates from 80,000 to 200,000. We set k as 10 and keep the other parameters the same as above. The comparative results between PKNN-DF, PKNN-BF and BJKS-KNN as shown in Fig. 24 are

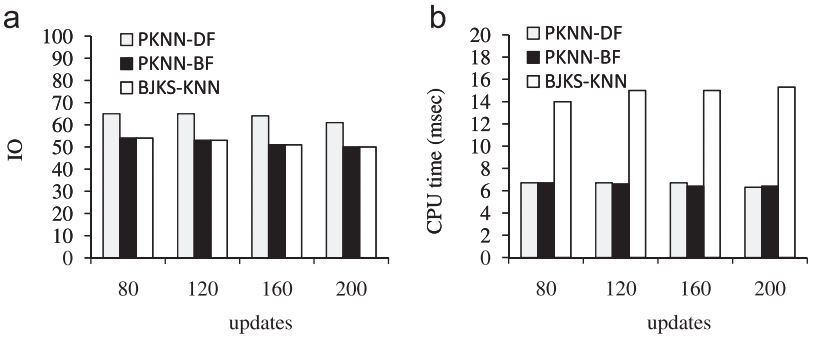

Fig. 24. KNN queries, varying updates: (a) page accesses and (b) CPU time.

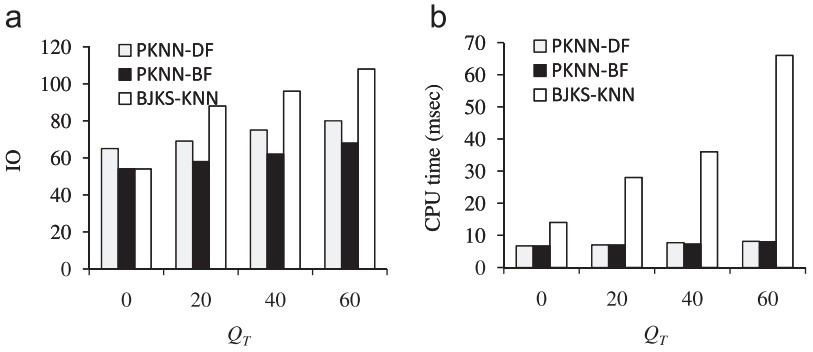

Fig. 25. KNN queries, varying Q_T: (a) page accesses and (b) CPU time.

similar to those in Fig. 23. We do not observe much performance change as the number of updates increases.

In the third experiment, we see how the querying period length affects the performance. We vary the length of the querying period Q_T from 0 to 60. We use the dataset with 80,000 updates while the other parameters remain the same as above. The I/O and CPU costs as functions of Q_T are plotted in Fig. 25. The relative performance between PKNN-DF, PKNN-BF and BJKS-KNN are similar to the previous results except a few differences. First, the difference between them become larger as the querying period length increases. This is because BJKS-KNN is a continuous kNN algorithm, which computes the kNN for all the timestamps in the querying period. As the querying period becomes longer, there are more different kNN result changes in the querying period and hence more processing costs. Another difference from the previous two sets of experiments is that BJKS-KNN is no longer optimal in terms of page access cost when the length of the querying period is non-zero. This is because BJKS-KNN finds the kNN set for any time point in the querying period, which results in both extra I/O and CPU cost. Therefore, BJKS-KNN has more I/Os than both PKNN-BF and PKNN-DF when the length of the querying period becomes non-zero. When the querying period is 60 time units long, the number of I/O of BJKS-KNN is about 50% more than those of PKNN-DF and PKNN-BF; the CPU time of BJKS-KNN is 8 times those of PKNN-DF and PKNN-BF. This confirms our statement that it is inefficient to use continuous kNN algorithms to process predictive kNN queries.

8.3. Evaluation of the cost model

In this subsection, we evaluate the accuracy of the cost model developed in Section 7.1 for predictive range queries. The queries are generated in the same way as before except that the radius is just one value Q_r rather than being distributed in the range $[0, Q_{Rmax}]$. This is because the cost model estimates the cost for a given query size. We still use the 100,000 objects dataset with 80,000 updates. In the first experiment, we vary Q_r from 2,500 to 10,000. The actual number of node accesses and the number computed by the model (Eq. 1) are shown in Fig. 26(a). The corresponding relative errors are shown in Fig. 26(b). We can see that the relative errors are all below 8%, which shows the high accuracy of our cost model.

In another experiment, we fix Q_r as 5000 but vary the size of the dataset from 25,000 objects to 100,000 objects. The other parameters of the datasets are the same as those for the dataset of 100,000 objects. The comparison between the actual and estimated numbers of nodes, as well as the relative errors are shown in Fig. 27. The errors are all below 10%. The accuracy gets higher as the dataset size gets larger. We have varied other parameters and done similar experiments. The relative errors are all less than 10%.

9. Conclusions and discussions

In this article, we introduced the *Transformed Minkowski Sum* (TMS) technique for determining the

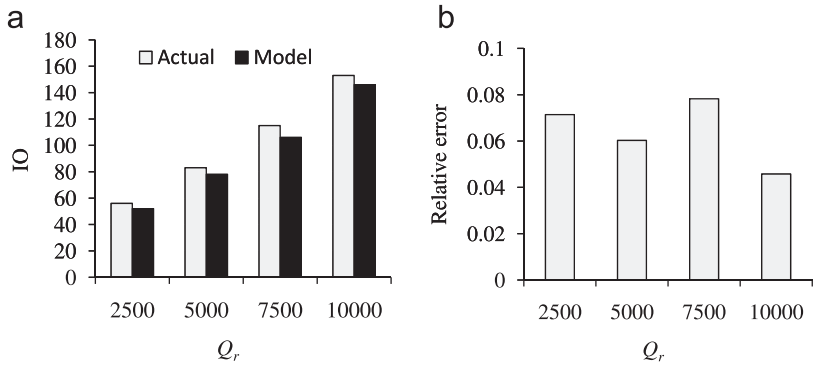

Fig. 26. Cost model, varying Q_r: (a) actual vs. model and (b) relative error.

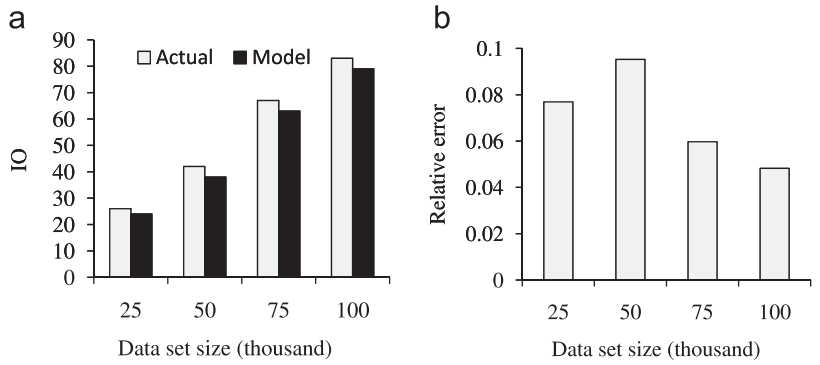

Fig. 27. Cost model, varying dataset size: (a) actual vs. model and (b) relative error.

intersection of two moving objects of arbitrary shapes. We have exemplified the usage of this technique by applying it to two typical types of queries on moving objects, that is, the predictive range and kNN queries. Based on the TMS, we proposed an I/O optimal algorithm for the predictive range query, which can return exactly the objects that intersect a moving circular region; this has not been achieved by any previous algorithm. We also proposed an I/O optimal algorithm for the predictive kNN query based on the TMS. Furthermore, we derived a cost model based on the TMS for estimating the number of node accesses of predictive range queries and showed how this model can help estimate the cost of predictive kNN queries. Through experiments, we have verified the effectiveness of the technique and shown that the proposed algorithms outperform competitive techniques in both I/O and CPU costs. The experiments also showed that our cost model has high accuracy.

Since the objects trajectories are represented as linear functions of time, it is possible to compute the shortest distance of objects and a range query through solving an equation system (note that it is not that trivial since there is a time range which causes complexities in boundary cases). Then we may use this as the pruning distance in the tree traversal to have an I/O optimal algorithm. A full investigation of this approach is deferred as future work. However, the TMS method is still preferable as a convenient, intuitive and generic tool for analyzing moving spatial objects. Moreover, it provides highly efficient algorithms which mostly need only a few inequality checks.

Acknowledgments

We would like to thank the authors of [4] for providing us the source code of their work, which we used in our performance study. This work is supported under the Australian Research Councils' Discovery funding schemes (Project number DP0880215 and DP0880250).

Appendix A. Lemma 2

Given a rectangle R and a circle S, the closest distance between R and S, is the closest distance between the Minkowski enlargement (sum) of R with regard to S and the center of S.

Proof. (A) If R intersects S:

According to Lemma 1, the Minkowski sum (we omit "of R and S" when the meaning is clear) intersects the center of S. The closest distance between R and S is 0. The closest distance between the Minkowski sum and the center of S is also 0.

(B) If R does not intersect S:

The Minkowski sum does not intersect the center of S. We extend the four sides of R and divide the plane outside

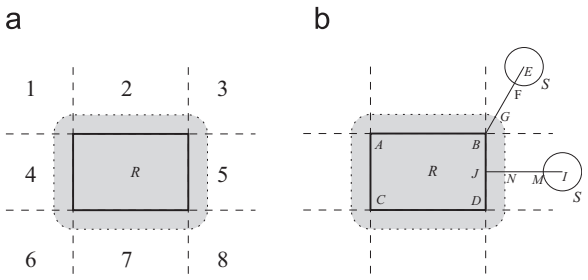

Fig. 28. Lemma 2.

R into eight regions as shown in Fig. 28(a). There are two scenarios as follow.

(B.1) If the center of S is in regions 1, 3, 6 and 8:

For example, the center of S is at point E as shown in Fig. 28(b). The closest corner of R to E is B. Line BE intersects S at F and the boundary of the Minkowski sum at G. It is easy to show that: the closest distance between R and S is $|BF|$; the closest distance between the Minkowski sum and E is $|EG|$. $|EF|$ equals $|BG|$, so $|BF|$ equals $|EG|$.

(B.2) If the center of S is in regions 2, 4, 5 and 7:

For example, the center of S is at point I as shown in Fig. 28(b). The closest side of R to I is BD. We can draw a line perpendicular to BD from I, intersecting BD at point J. Line IJ intersects S at M and the boundary of the Minkowski sum at N. It is easy to show that: the closest distance between R and S is $|JM|$; the closest distance between the Minkowski sum and I is $|IN|$. $|IM|$ equals $|JN|$, so $|JM|$ equals $|IN|$. □

References

[1] P.K. Agarwal, L. Arge, J. Erickson, Indexing moving points, in: PODS2000.
[2] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: SIGMOD1990.
[3] R. Benetis, C.S. Jensen, G. Karciauskas, S. Saltenis, Nearest neighbor and reverse nearest neighbor queries for moving objects, in: IDEAS, 2002, pp. 44–53.
[4] R. Benetis, C.S. Jensen, G. Karciauskas, S. Saltenis, Nearest and reverse nearest neighbor queries for moving objectsVLDB Journal 15 (3) (2006) 229–249.
[5] C. Bohm, A cost model for query processing in high-dimensional data spacesTODS 25 (2) (2000) 129–178.
[6] S Chen, C.S. Jensen, D. Lin, A benchmark for evaluating moving object indexesPVLDB 1 (2) (2008) 1574–1585.
[7] S. Chen, B. Chin Ooi, K.-L. Tan, M.A. Nascimento, St^{2}b-tree: a self-tunable spatio-temporal b$^+$-tree index for moving objects, in: SIGMOD Conference, 2008, pp. 29–42.
[8] G. Ewald, Combinatorial Convexity and Algebraic Geometry, Springer, 1996.
[9] G.R. Hjaltason, H. Samet, Ranking in spatial databases, in: SSD1995.
[10] H. Hu, J. Xu, D.L. Lee, A generic framework for monitoring continuous spatial queries over moving objects, in: SIGMOD Conference, 2005, pp. 479–490.
[11] G.S. Iwerks, H. Samet, K.P. Smith, Continuous k-nearest neighbor queries for continuously moving points with updates, in: VLDB, 2003, pp. 512–523.
[12] H.V. Jagadish, A. Silberschatz, S. Sudarshan, Recovering from main-memory lapses, in: VLDB, 1993, pp. 391–404.
[13] C. Jensen, D. Lin, B.C. Ooi, Query and update efficient B+-tree based indexing of moving objects, in: VLDB2004.
[14] G. Kollios, D. Gunopulos, V.J. Tsotras, On indexing mobile objects, in: PODS1999.
[15] K. Mouratidis, M. Hadjieleftheriou, D. Papadias, Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring, in: SIGMOD Conference, 2005, pp. 634–645.
[16] J.M. Patel, Y. Chen, V.P. Chakka, STRIPES: an efficient index for predicted trajectories, in: SIGMOD2004.
[17] K. Raptopoulou, A. Papadopoulos, Y. Manolopoulos, Fast nearest-neighbor query processing in moving-object databasesGeoInformatica 7 (2) (2003) 113–137.
[18] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: SIGMOD1995.
[19] S. Saltenis, C.S. Jensen, S.T. Leutenegger, M.A. Lopez, Indexing the positions of continuously moving objects, in: SIGMOD2000.
[20] H. Samet, The quadtree and related hierarchical data structuresACM Computing Survey 16 (2) (1984) 187–260.
[21] Y.N. Silva, X. Xiong, W.G. Aref, The rum-tree: supporting frequent updates in r-trees using memosVLDB Journal 18 (3) (2009) 719–738.
[22] A.P. Sistla, O. Wolfson, S. Chamberlain, S. Dao, Modeling and querying moving objects, in: ICDE1997.
[23] Y. Tao, D. Papadias, Spatial queries in dynamic environmentsTODS 28 (2) (2003) 101–139.
[24] Y. Tao, D. Papadias, J. Sun, The TPR*-tree: an optimized spatio-temporal access method for predictive queries, in: VLDB2003.
[25] Y. Tao, J. Sun, D. Papadias, Analysis of predictive spatio-temporal queriesTODS 28 (4) (2003) 295–336.
[26] Y. Tao, X. Xiao, Primal or dual: which promises faster spatiotemporal search?VLDB Journal 17 (5) (2008) 1253–1270
[27] Y. Tao, J. Zhang, D. Papadias, N. Mamoulis, An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spacesIEEE Transactions on Knowledge and Data Engineering 16 (10) (2004) 1169–1184.
[28] X. Xiong, M.F. Mokbel, W.G. Aref, Sea-cnn: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases, in: ICDE, 2005, pp. 643–654.
[29] M.L. Yiu, Y. Tao, N. Mamoulis, The^{ual}-tree: indexing moving objects by space filling curves in the dual spaceVLDB Journal 17 (3) (2008) 379–400.
[30] X. Yu, K.Q. Pu, N. Koudas, Monitoring k-nearest neighbor queries over moving objects, in: ICDE, 2005, pp. 631–642.
[31] R. Zhang, D. Lin, K. Ramamohanarao, E. Bertino, Continuous intersection joins over moving objects, in: ICDE, 2008, pp. 863–872.