Continuous Visible k Nearest Neighbor Query on Moving Objects

Yanqiu Wang[a], Rui Zhang[b], Chuanfei Xu[a], Jianzhong Qi[b], Yu Gu[a], Ge Yu[a,*]

[a] *Department of Computer Software and Theory, Northeastern University, China.*
[b] *Department of Computing and Information Systems, University of Melbourne, Australia.*

Abstract

A visible k nearest neighbor (VkNN) query retrieves k objects that are visible and nearest to the query object, where "visible" means that there is no obstacle between an object and the query object. Existing studies on the VkNN query have focused on static data objects. In this paper we investigate how to process the query on moving objects continuously. We propose an effective filtering-and-refinement framework for evaluating this type of queries. We exploit spatial proximity and visibility properties between the query object and data objects to prune search space under this framework. A detailed cost analysis and a comprehensive experimental study are conducted on the proposed framework. The results validate the effectiveness of the pruning techniques and verify the efficiency of the proposed framework. The proposed framework outperforms a straightforward solution by an order of magnitude in terms of both communication and computation costs.

Keywords: Spatio-temporal databases, continuous visible k nearest neighbor queries, safe region, invisible time period

1. Introduction

The *visible k nearest neighbor (VkNN)* query has attracted great research interest [12, 11, 19, 20] recently due to emerging applications such as security camera placement and sightseeing site recommendation. This query assumes a set $\mathcal{P}$ of data objects, a set O of obstacles (represented by line segments) and a query object q. Then it retrieves k data objects from $\mathcal{P}$ that are visible and nearest to q. Figure 1 gives an example. Suppose $k = 2$. The data objects are listed according to their Euclidean distance to q as: $p_5, p_6, p_4, p_3, p_1, p_2$. Since p_5, p_4 and p_1 are blocked by the obstacles and invisible to q, they are not answers to the VkNN query. The VkNN set of q, denoted by $VkNN(q)$, is $\{p_6, p_3\}$.

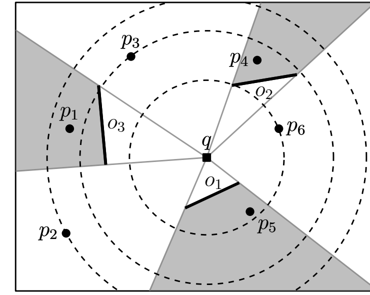


Figure 1: $VkNN(q) = \{p_6, p_3, p_2\}$ ($k = 2$)

In this paper we study a continuous version of the VkNN query, namely, the *continuous VkNN query*, which computes the VkNN from a set of moving objects for a moving query object continuously (i.e., for every timestamp).

The continuous VkNN query has various applications. For example, in a military simulation, there can be more than 100,000 moving objects [35] such as soldiers and military vehicles interacting with each other. A soldier needs to keep track of his/her nearest visible enemies, so that he/she can attack or avoid them. As the soldier and the enemies are moving constantly, the simulator needs to monitor them continuously and report to the soldier his/her nearest visible enemies. In another example, massively multiplayer online first-person shooter (MMOFPS) games like CrossFire[1] need to show a player his/her nearby visible players so that he/she can shoot them. Again, as the players are moving constantly, the game system needs to monitor the players continuously and report to the player his/her nearest visible players. There may be millions of players [9] online at the same time. Therefore, a highly efficient algorithm is required to provide nearest-visible-player monitoring in real time.

The continuous VkNN query is interesting not only for its real applications but also for the technical challenges it raises. To the best of our knowledge, there is no existing work considering the query on moving objects. The main challenge here is that the query result needs to be up-to-date at every timestamp, which incurs significant communication and computation costs. To mitigate the costs, we propose a filtering-and-refinement query processing framework, and exploit spatial proximity and visibility properties between the query object and the data objects to prune search space under the framework. For spatial proximity based pruning, we use the *safe region*, which is a circular region centered at an object and is defined to bound the

*Corresponding author. Tel.: +86 2483683113
Email addresses: wangyanqiu@ise.neu.edu.cn (Yanqiu Wang), rui@csse.unimelb.edu.au (Rui Zhang), chuanfeixu@research.neu.edu.cn (Chuanfei Xu), jiqi@csse.unimelb.edu.au (Jianzhong Qi), guyu@ise.neu.edu.cn (Yu Gu), yuge@ise.neu.edu.cn (Ge Yu)

[1] http://crossfire.z8games.com/

movement of the object for a certain period of time T (cf. Figure 2). The safe regions that are close and visible to the query object further define a pruning region, which can be used to rule out objects that are too far away to be in the VkNN set within T timestamps. For objects that survive the safe region based pruning, their distance to the query object is not too far, but they may still be invisible to the query object due to obstacles. This motivates the visibility based pruning, which utilizes sub-periods within T that an object is invisible to the query object, so that the object can be excluded from the VkNN candidates during those sub-periods. We call such sub-periods the *invisible time periods*. All pruning techniques together keep the number of objects that pass the filtering stage small, and hence substantially reduce the costs of the refinement stage. As a result, we achieve a highly efficient query processing framework.

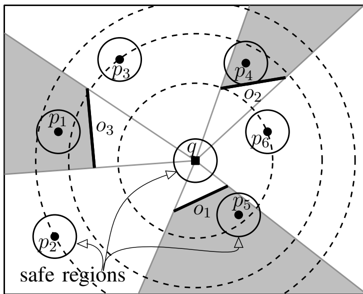


Figure 2: Safe regions

We summarize the contributions of this paper as follows.

- This is the first study that addresses the continuous VkNN query on moving objects. We propose a filtering-and-refinement framework that can process the query effectively.
- We develop two pruning strategies, namely, safe region based pruning and invisible time period based pruning, to reduce the search space for query processing under the proposed framework.
- We conduct a detailed cost analysis for the proposed pruning techniques. Extensive experiments using both real and synthetic data sets demonstrate the high efficiency of the pruning techniques as well as the proposed query processing framework.

The rest of this paper is structured as follows. We first review related work in Section 2. Then we formalize the continuous VkNN query on moving objects and present the filtering-and-refinement framework in Section 3. In Section 4, we present two pruning strategies under the framework and in Section 5 we provide a cost analysis for algorithms based on these pruning strategies. We report the experimental results in Section 6 and conclude the paper in Section 7.

2. Related Work

We review three classes of related studies, namely, continuous spatial queries in general, continuous k nearest neighbor queries on moving objects and visible k nearest neighbor queries on static objects.

2.1. Continuous Spatial Queries in General

There is a large body of literature on continuous evaluation of spatial queries. For example, Šaltenis *et al.* [28] propose the Time Parameterized R-tree (TPR-tree) that indexes moving points as linear functions of time, based on which time-parameterized queries [25] are proposed to retrieve moving objects that satisfy certain time-parameterized predicates continuously. Hu *et al.* [15] propose a safe region based framework for monitoring continuous spatial queries over moving objects on a client-sever based system. Each moving object (a client) in the system is aware of the current query result and only reports its new location to the server if it is likely to cause changes to the query result. Mokbel *et al.* [16, 17] propose two frameworks for processing continuous spatial queries. They process the queries incrementally by computing the effect of each individual update on the query answer. They also propose shared execution techniques to process multiple queries at the same time. Benetics *et al.* [6] study the problem of continuous reverse nearest neighbor (RNN) monitoring over moving objects. Xia *et al.* [30] also study the continuous RNN query. They propose a so-called six-region approach to process the query. Later, Cheema *et al.* [8] propose a safe region based approach to process the continuous reverse k nearest neighbor (RkNN) query. More recently, Zhang *et al.* [35] study the continuous intersection join query, which reports the set of intersecting objects from two moving object sets continuously.

2.2. Continuous k Nearest Neighbor Queries on Moving Objects

As a major type of continuous spatial queries, the continuous k nearest neighbor (kNN) query on moving objects has been studied extensively. For example, Tao *et al.* [26] consider a continuous NN search that retrieves the NNs of every point on a given line segment. Song *et al.* [24] propose a sampling-based approach to reduce the cost of processing continuous NN queries. However, this method may produce inaccurate query answers. Zhang *et al.* [33] use the validity region to enable mobile clients to determine the validity of previous NN query results based on their current locations. Nutanong *et al.* [21, 22] use an incremental-safe-region based technique called the V*-Diagram that exploits both the location of the query object and the scope of the current search space to answer moving kNN queries. Yu *et al.* [32] use grid indices for monitoring kNN queries on moving objects, which can provide exact query answers but with a delay in the time. Xiong *et at.* [31] also use grid indices for monitoring continuous kNN queries. They compute the query answer incrementally by maintaining an answer region for each query. The location updates of objects at each timestamp are used to update the answer region, which is then used to generate updates to the query answer. In addition, Mouratidis *et al.* [18] propose a threshold based algorithm for the continuous NN query, which aims at minimizing the communication overhead between the query processor and the moving objects. Further more, Hsueh *et al.* [14] present a partition

based lazy update algorithm that uses Location Information Tables and safe regions to process continuous NN queries.

There are also many studies on variants of the continuous kNN query. For example, Zhang *et al.* [34] study the predictive moving kNN query. Hashem *et al.* [13] study how to protect a user's location privacy in moving kNN monitoring systems. Shahabi and Sharifzadeh [23] propose the VoR-Tree, which incorporates Voronoi diagrams into the R-tree to improve the efficiency of processing various types of kNN queries. Al-Amri *et al.* [1, 3] study kNN queries in indoor space. They propose an adjacency index structure for moving objects in indoor space that takes into account both spatial and temporal properties. They use a non-leaf node timestamping method to store moving data and process kNN queries. In addition, Al-Amri *et al.* [2] propose a moving object index and a lookup table that can be used to process traditional kNN queries as well as the novel direction and velocity queries. Since these studies do not consider obstacles in the continuous kNN query, their methods do not apply to our problem.

2.3. Visible k Nearest Neighbor Queries on Static Objects

The visible k nearest neighbor (VkNN) query is first proposed by Nutanong *et al.* [19]. This query retrieves the nearest k static objects that are visible to a static query object. To process the query, Nutanong *et al.* [19] propose an algorithm that starts from retrieving the nearest object and then incrementally obtains the knowledge of visibility while finding other nearest neighbors. Nutanong *et al.* [20] also study the aggregate VkNN query and propose an approach named single retrieval front to process the query. Besides, Gao *et al.* [10] study the visible reverse k nearest neighbor (VRkNN) query, which finds all objects regarding the query object as a member of their VkNN sets. In another paper [12], Gao *et at.* study the continuous visible nearest neighbor (CVNN) query. They assume static data objects and a query object q moving along a given line segment, and use a branch-and-bound technique to process the CVNN query based on 7 pruning heuristics. Among the 7 pruning heuristics, 4 are based on the distance between the *static* data objects and the line segment where q moves along, while the other 3 are based on the position relationship between the *static* data objects, an obstacle and q. Since we do not assume a line segment for q or static data objects, these pruning heuristics do not apply.

3. Preliminaries

In this section, we formulate the continuous VkNN query on moving objects and present an efficient filtering-and-refinement framework to process the query. We summarize the symbols frequently used in the following discussion in Table 1.

3.1. Problem Formulation

We first define the concept of *visibility* between two objects. We assume that obstacles are line segments while data objects including the query object are points in a two-dimensional Euclidean space.

Definition 1. *(**Visibility**). Given a set O of obstacles, we say that two objects p and q are visible to each other iff there is no obstacle o in O such that the line segment connecting p and q, denoted by $\overline{pq}$, intersects o,* i.e., $\forall o \in O, o \cap \overline{pq} = \emptyset$.

Similarly, we say that p and q are invisible to each other iff there is an obstacle o such that $\overline{pq}$ intersects o, i.e., $\exists o \in O, o \cap \overline{pq} \neq \emptyset$.

We use $\mathcal{V}(q)$ to denote the *visible region* of q, i.e., the region inside which all points are visible to q (cf. the white region in Figure 2). By checking whether p resides in $\mathcal{V}(q)$, we know whether p is visible to q. Similarly, we use $\mathcal{I}(q)$ to denote the *invisible region* of q inside which all points are invisible to q (cf. the grey region in Figure 2). We follow the classical technique to compute $\mathcal{V}(q)$ [4, 27]. The technique uses the obstacles and the lines that pass through q and the endpoints of the obstacles to clip the invisible region of q from the whole data space. The remaining part of the data space is the visible region of q.

We define the *visible distance (VD)* to represent the distance between two objects when their visibility is taken into consideration.

$$VD(p,q) = \begin{cases} dist(p,q) & p \text{ is visible to } q \\ \infty & \text{otherwise.} \end{cases}$$

Here, $dist(p, q)$ denotes the Euclidean distance between two points p and q.

Given the definition of visible distance, we define the *continuous visible k nearest neighbor query on moving objects* as follows.

Definition 2. *(**Continuous Visible** k **Nearest Neighbor (CV*k*NN) query on moving objects**) Given a set $\mathcal{P}$ of moving objects, a set O of obstacles and a moving query object q, at every timestamp, the continuous visible k nearest neighbor query retrieves a subset of $\mathcal{P}$, denoted by $VkNN(q)$, such that:*

(i) $|VkNN(q)| = k$;

(ii) $\forall p \in VkNN(q)$, p is visible to q;

(iii) $\forall p \in VkNN(q)$ and $\forall p' \in \mathcal{P} \setminus VkNN(q)$, $VD(p,q) \leq VD(p',q)$.

System architecture. We consider processing the CVkNN query for two types of systems, centralized systems and client-server based systems. Centralized systems are commonly used in applications such as military simulations, where the simulation system usually controls the moving objects directly and has all up-to-date object location information it needs to process the CVkNN query. Client-server based systems are commonly used in applications such as multiplayer online games, where most of the moving objects are avatars controlled by human players through game clients (local computers or play stations), while the CVkNN query is processed in the game server. In this case, communication cost is incurred when the server and the clients exchange location updates and query results.

For the algorithm design and cost analysis in the rest of this paper, we assume a client-server based system for ease of presentation, i.e., we will consider computation cost as well as communication cost. However, the algorithm design principles and cost analysis also apply to centralized systems, in which

Symbol	Definition
$\mathcal{P}$	a set of moving objects
O	a set of obstacles (line segments)
p	a moving object
o	an obstacle
q	a query object
$\overline{pq}$	a line segment that connects p and q
$\lvert\overline{pq}\rvert$	the length of $\overline{pq}$ (the Euclidean distance between p and q)
$VD(p, q)$	the visible distance between p and q
R_p	the safe region of p
R_q	the safe region of q
$\mathcal{V}(q)$	the visible region of q
$\mathcal{V}(R_q)$	the visible region of R_q
$\mathcal{I}(q)$	the invisible region of q
$\mathcal{I}(R_q)$	the invisible region of R_q
$MinVD(R_q, R_p)$	the minimum visible distance between R_q and R_p
$MaxVD(R_q, R_p)$	the maximum visible distance between R_q and R_p
S_c	a set of query answer candidate objects
v_m	the global maximum speed
T	the recomputation period of S_c
τ_p	an invisible time period of p
τ_p^l	an invisible time period lower bound of p
τ_p^m	a moving direction aware invisible time period of p

Table 1: Frequently Used Symbols

case communication cost is replaced by the cost of accessing the object location data within the centralized systems.

We also assume that the CVkNN query is processed in main memory. This is reasonable because storing a data set of 100,000 objects only takes several MB while currently a commodity computer usually has several GB of main memory.

3.2. Query Processing Framework

A straightforward solution to the CVkNN query is to perform a *snapshot VkNN query at every timestamp* as follows. At every timestamp, we solicit location updates for all moving objects, sort the objects based on their visible distance to the query object, and then report the first k objects as the query answer. This will serve as our baseline algorithm since there is no existing work on the CVkNN query. We might use the solution proposed by Nutanong *et al.* [19] to process a snapshot VkNN query, but this solution is not efficient under our problem settings for the following reason. Nutanong *et al.* assume an R-tree on the data objects and the obstacles, and then use a best-first traversal to identify the invisible regions and the query object's nearest visible objects gradually. Applying their method means building an R-tree on the objects and the obstacles at every timestamp, which has a time complexity of $O((|\mathcal{P}|+|O|)\log(|\mathcal{P}|+|O|))$. In our problem settings, the obstacles are static. We just need to build an R-tree to index them once. Then computing the visible distance and sorting the moving objects based on visible distance at every timestamp requires only $O(|\mathcal{P}|\log|\mathcal{P}|)$ time, which is smaller than that of Nutanong *et al.*'s algorithm. Therefore, we use the sorting based snapshot VkNN query algorithm instead of Nutanong *et al.*'s algorithm as the baseline. However, this baseline solution still requires too much communication and computation costs incurred by soliciting location updates and sorting for all objects at every timestamp. The need for a more efficient solution is evident.

We propose a filtering-and-refinement framework to process the CVkNN query on moving objects. The framework first performs a two-stage filtering to reduce the search space. Then refinement is performed to examine the exact positions of the unfiltered objects and find the query result.

As Figure 3 shows, we divide the time axis into periods of T timestamps each, where T is a system parameter. At the beginning of each period, we solicit location updates for all objects, which means that we only need to solicit location updates for all objects once every T timestamps. We will analyze the effect of T on system performance in Section 4.1 and perform an empirical study in Section 6.2. After the location updates are solicited, we build an R-tree on the moving objects (safe regions of the objects, actually). At every T timestamps, we update the R-tree nodes according to the new locations of the objects. We use the R-tree to perform the first stage filtering, which results in a subset of $\mathcal{P}$ that is *guaranteed to contain all possible VkNNs* of the query object for the next T timestamps. We call this subset the candidate answer set and denote it by S_c. During the next T timestamps, we perform the second stage filtering on S_c to determine the objects that require to be examined by refinement. This filtering stage is based on the visibility relationship between the object and *no false dismissal will be introduced*. We then sort the unfiltered objects based on their visible distance to the query object to generate the *exact* query result at each timestamp. This process repeats and answers are reported continuously for the CVkNN query.

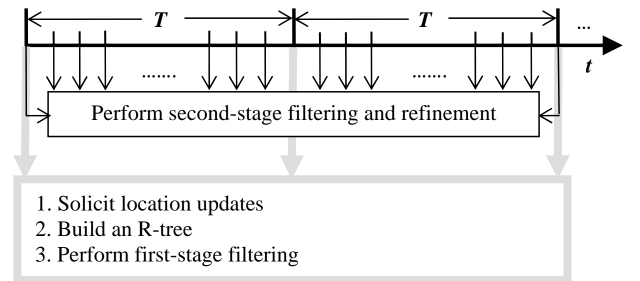


Figure 3: The query processing framework

Discussion. The advantage of our framework is that it only needs to solicit location updates for all objects every T timestamps, while the snapshot VkNN based solution requires doing that at every timestamp. A possible concern is that our framework might have a high processing cost at the beginning of every period incurred by the three operations, i.e., (i) soliciting location updates, (ii) building an R-tree, and (iii) filtering the search space. We argue that these operations are only performed once every T timestamps and hence the cost

is amortized. Further, operations (i) and (iii), soliciting location updates and filtering the search space, are required for any algorithm that may efficiently process the CVkNN query anyway. As for operation (ii), building an in-memory R-tree is efficient [7, 29]. Our experimental study shows that building an in-memory R-tree of size 10,000 takes only about 0.33 seconds. This small overhead enables effective R-tree based pruning that significantly reduces the number of data objects to be checked in the refinement stage. Therefore, the cost of building an R-tree once every T timestamps is justified. We also consider maintaining the R-tree incrementally, i.e., we build an R-tree at start and update it as the objects move. The problem of the incremental maintenance is that it has to process every object update, which is too expensive when the objects update frequently. As our experiments show, rebuilding the R-tree is much more efficient than maintaining it incrementally under our settings.

Processing multiple queries concurrently. When there are multiple concurrent CVkNN queries, we can process them together and reduce the processing costs by shared execution. In particular, in the first stage filtering, the R-tree on the moving objects is shared by multiple CVkNN queries to compute a candidate answer set for each query, which can be done by a grouped branch-and-bound search on the R-tree. This way we constrain the communication and computation costs. In the second stage filtering, we examine each candidate answer set based on the visibility relationship to obtain the VkNNs at each timestamp. This filtering stage requires soliciting the locations of some objects in the candidate answer sets. Since an object may belong to multiple candidate answer sets, its solicited location can be shared by the filtering of multiple candidate answer sets. This way we reduce the communication cost.

4. Pruning Techniques

Our query processing framework has two filtering stages. In the first filtering stage, the aim is to generate a set of query answer candidates S_c that can stay valid for the next T timestamps while the size of the set is as small as possible. In the second filtering stage, the aim is to further limit the number of objects in S_c that need to be examined at each timestamp by the refinement stage. Towards these aims, we propose a safe region based pruning method and an invisible time period based pruning method summarized as follows:

- **Safe Region based Pruning.** We use the *safe region* to define a region where a moving object must be in during a period of T timestamps (cf. Section 4.1). Then for the query object's safe region, we find its kth nearest data object's safe region that is entirely visible to it. The distance between these two safe regions defines a region where an object's safe region must intersect or be enclosed in so that this object can be in the query answer candidate set S_c. All other objects are pruned from further processing.

- **Invisible Time Period based Pruning.** The objects in S_c are close to the query object q but may not be visible to q throughout a period of T timestamps. We call the subperiod when an object p ($p \in S_c$) is invisible to q the *invisible time period* of p. Since an object's exact movement is not predictable, there is no way to compute an exact invisible time period. Instead, we compute a *lower bound* of the invisible time period of p based on the current positions of p, q and the obstacles between the two objects. Then during the bounded period, p can be excluded from S_c and need not to be examined in the refinement stage. When the bounded period expires, we solicit a location update of p and compute the next invisible time period lower bound for p. This process repeats for the objects in S_c. Since this makes some objects not be examined at every timestamp, the cost of the refinement stage is reduced.

  To further improve the pruning capability of the invisible time period based method, we utilize the query object's moving direction for the computation of the invisible time period lower bound, which results in the *moving direction aware invisible time period* that is generally longer and always not shorter than the basic invisible time period lower bound.

Next we elaborate the pruning techniques.

4.1. Safe Region based Pruning

We assume a global maximum speed of all objects, denoted by v_m, which may be the greatest speed limit or the speed of the fastest object in a gaming/simulator system. Then during T timestamps, the movement of an object p must be within a circular region centered at p with $v_m \cdot T$ being the radius. We call this circular region the safe region of p (cf. Figure 4).

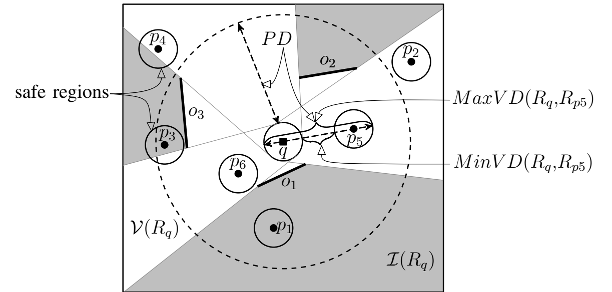


Figure 4: Safe region based pruning ($k = 2$)

4.1.1. The Pruning Distances

As objects' safe regions bound their movement, objects whose safe regions are not close enough to the query object's safe region can be discarded from the query answer candidate set S_c. We define three distances on the safe regions, which will then be used for the safe region based pruning.

We first extend the visible region of an object to the visible region of a region: the *visible region of a region* R, denoted by $\mathcal{V}(R)$, is defined as the intersection of the visible regions of all points in R, i.e., a point in $\mathcal{V}(R)$ is visible to every point in R. Similarly, the *invisible region* of R, denoted by $\mathcal{I}(R)$, is defined as the union of the invisible regions of all points in R. For example, in Figure 4, the white region is the visible region

of R_q, $\mathcal{V}(R_q)$, while the gray region is the invisible region of R_q, $\mathcal{I}(R_q)$. By definition, if an object is in $\mathcal{V}(R_q)$, it is guaranteed that the object will be visible to q for the following T timestamps. To compute $\mathcal{V}(R_q)$, we draw tangent lines to R_q from the endpoints of the obstacles in O. The tangent lines and the obstacles clip the invisible region from the data space. The remaining region forms $\mathcal{V}(R_q)$ (cf. Figure 4). Once we have $\mathcal{V}(R_q)$, checking whether a region is (entirely) visible to R_q is done by checking whether the region is (fully) overlapped by $\mathcal{V}(R_q)$.

Now we can define the minimum visible distance, the maximum visible distance, and the pruning distance. The *minimum visible distance* (*MinVD*) between R_q and a region R, denoted by $MinVD(R_q, R)$, is defined as the smallest distance between a point in R_q and a point from R that is visible to R_q (i.e., in the visible region of R_q). Formally,

$$MinVD(R_q, R) = \begin{cases} MinDist(R_q, R \cap \mathcal{V}(R_q)) & R \text{ is visible to } R_q \\ \infty & \text{otherwise.} \end{cases}$$

Here, $MinDist(\cdot)$ is a function that returns the smallest distance between any two points from two regions.

Similarly, we define the *maximum visible distance* (*MaxVD*) between R_q and a region R, denoted by $MaxVD(R_q, R)$, to be the largest distance between a point in R_q and a point from R that is visible to R_q. Formally,

$$MaxVD(R_q, R) = \begin{cases} MaxDist(R_q, R \cap \mathcal{V}(R_q)) & R \text{ is visible to } R_q \\ \infty & \text{otherwise.} \end{cases}$$

Here, $MaxDist(\cdot)$ is a function that returns the largest distance between any two points from two regions.

The *pruning distance (PD)* is then defined as the largest *MaxVD* of the k nearest safe regions who are entirely visible to R_q. Formally,

$$\begin{aligned} &PD = \max\{MaxVD(R_q, R) | R \in S_R\}, S_R \text{ satisfies} \\ &\text{(i) } |S_R| = k \\ &\text{(ii) } \forall R \in S_R\ \forall R' \notin S_R,\ R \cap \mathcal{V}(R_q) = R \text{ and } R' \cap \mathcal{V}(R_q) = R' \\ &MinVD(R_q, R) < MinVD(R_q, R'). \end{aligned}$$

The definition of PD guarantees that, during the following T timestamps, we have k objects that are always visible to q, while their distance to q is at most PD. Any object whose safe region has a MinVD to R_q that is larger than PD cannot contribute to the query answer candidate set S_c (e.g., p_2 in Figure 4). Therefore, using PD for pruning guarantees no false dismissal.

4.1.2. The Pruning Algorithm

Safe region based pruning works as follows. We solicit object location updates at the beginning of every period (i.e., every T timestamps) and update the R-tree $T_{\mathcal{P}}$. Then, we traverse $T_{\mathcal{P}}$ in a best-first order to determine S_c for the next T timestamps. We start the traversal with inserting all entries of the root node of $T_{\mathcal{P}}$ into a priority queue $Q_{\mathcal{P}}$. The entries in $Q_{\mathcal{P}}$ are prioritized based on their minimum visible distance to the safe region of the query object q, denoted by R_q. They are popped out one after another until $Q_{\mathcal{P}}$ is empty. When an entry e_p pops out, if it is *invisible* to R_q, which means any point bounded by e_p is *invisible* to every point in R_q, then we simply discard the entry. Otherwise, (i) if e_p has a child node, then all entries in the child node are inserted into $Q_{\mathcal{P}}$; (ii) if it is a data entry, which represents the safe region of a data object p, denoted by R_p, then we add p to S_c. We further check whether R_p is *entirely visible* to R_q, which means any point in R_p is *visible* to every point in R_q. If it is, then p must be visible to q and be a VkNN candidate throughout the next T timestamps. We repeat the above process until we have found k data objects whose safe regions are entirely visible to R_q. These k objects guarantee that we have a candidate VkNN set for q for the next T timestamp period. We compute the pruning distance PD to define a circular region and bound all these k objects' safe regions. If there is an object p whose safe region's minimum visible distance is larger than this pruning distance, then p cannot be closer to q than any of these k objects. Therefore, any entry popped out from $Q_{\mathcal{P}}$ whose minimum visible distance to R_q is larger than PD can be discarded from S_c and hence we reduce the search space. We summarize the pruning algorithm in Algorithm 1, where T_O denotes an R-tree that indexes the set of obstacles O to facilitate the computation of visibility relationships.

Algorithm 1: Safe Region based Pruning

```
Input  : R_q, T_P, T_O, k
Output: Candidate set S_c
1  Initialize Q_P with the entries in the root node of T_P;
2  PD ← ∞, S_c ← ∅;
3  while NOT Q_P.empty() do
4      e_p ← Q_P.pop();
5      if MinVD(R_q, e_p) > PD then
6          break;
7      if e_p is visible to R_q then
8          if e_p has a child node then
9              Insert all entries in the child node of e_p into
               Q_P;
10         else
11             S_c ← S_c ∪ p;
12             if there are k objects in S_c that are entirely
               visible to R_q then
13                 Update PD;
14
15 Return S_c.
```

Figure 4 gives an example to the above algorithm where we compute the CV2NN. We find 2 objects p_5 and p_6 whose safe regions are entirely visible to the safe region of q. They must be in S_c for the next T timestamps. Between these two objects, p_5 has a larger $MaxVD$ value. Therefore, $PD = MaxVD(R_q, R_{p5})$. Further, since the entries are prioritized in $Q_{\mathcal{P}}$ based on $MinVD$, we can early terminate the algorithm once such an entry is popped out from $Q_{\mathcal{P}}$ and thus reduce the computation cost.

4.1.3. Choosing the Value of T

The value of T significantly affects the system performance. A larger T may reduce the cost of query processing by reducing

the number of times that the R-tree is rebuilt. However, it will also increase the size of the safe regions and hence the number of candidate answers to be checked during the second filtering stage. Determining the best value of T theoretically is too difficult if possible at all. It requires an accurate model to predict the cost of a moving kNN query with the presence of obstacles and for real data distribution and movement patterns. A full study on such a detailed cost model is beyond the scope of this study. Therefore, in our experimental study, we choose the best value of T empirically in Section 6.2.

In a moving object management system, the parameter T may be self-adjusted based on the statistics of the costs of rebuilding the R-tree and maintaining the candidate query answer set. When the cost of rebuilding the R-tree dominates, the parameter T should be adjusted to a larger value to reduce the cost. Otherwise, the parameter T should be adjusted to a smaller value.

4.2. Invisible Time Period based Pruning

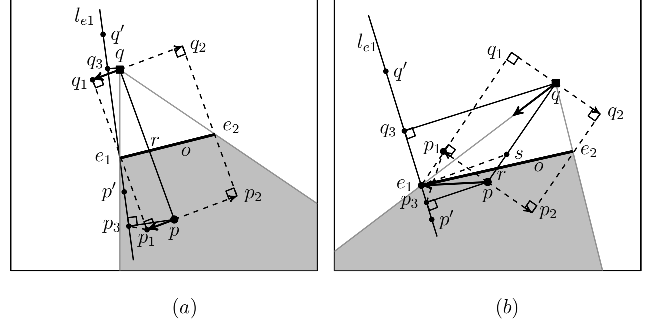
After the safe region based pruning, we have a set S_c that contains the possible VkNNs of the query object q for a period of T timestamps. The set S_c consists of two types of objects. *Type I* includes objects whose safe regions are entirely visible to R_q - these objects determine the pruning distance PD (cf. Figure 4, p_5 and p_6). *Type II* includes objects whose safe regions are only partially visible to R_q but their minimum visible distance is smaller than PD (cf. Figure 4, p_3). The latter type has the potential of further reducing the search space for VkNN computation. Specifically, a Type II object p needs time to move to be visible to q. We call this time the *invisible time period* of p, denoted by τ_p. If we could compute the value of τ_p, we could exclude p from S_c until τ_p expires. Unfortunately, we cannot predict the exact movement of p and thus, we cannot compute the exact value of τ_p. Instead, we compute a lower bound of τ_p that is guaranteed to expire before τ_p. Then we can exclude p from S_c until this lower bound expires, at which point we solicit a location update of p and check if p has actually became visible to q. (i) If yes then we add it back to S_c and continue with the regular refinement process. When p becomes invisible to q again, we recompute a lower bound of τ_p. (ii) Otherwise we directly recompute a lower bound of τ_p based on the updated location of p. By this means, we further reduce the size of S_c and hence reduce the query processing costs.

Next we explore how to derive a lower bound of τ_p. We first derive a basic lower bound, denoted by τ_p^l, and then refine it by taking the query object's moving direction into consideration, which results in an improve lower bound denoted by τ_p^m.

4.2.1. A Lower Bound of the Invisible Time Period

A lower bound estimation of τ_p, denoted by τ_p^l, must guarantee that p is invisible to q within τ_p^l, i.e., $\overline{pq} \cap o \neq \emptyset$, where o denotes an obstacle between p and q. A critical point is when $\overline{pq}$ reaches an endpoint of o, and the shortest time required for this to happen defines τ_p^l. Figure 5(a) gives an example, where $\overline{pq}$ needs to reach e_1 so that p and q can be visible to each other.

Assume that p and q can both move at the global maximum speed v_m towards arbitrary directions. We observe that, for $\overline{pq}$



(a) (b)

Figure 5: Computation of τ_p^l

to reach an endpoint of o with the shortest time, p and q should both move towards either a same endpoint of o or a same direction that is perpendicular to the line segment that connects the original locations of p and q, depending on which way results in the shortest moving time. For example, in Figure 5(a), there are four choices of movement, i.e., $\{p \to e_1, q \to e_1\}$, $\{p \to e_2, q \to e_2\}$, $\{p \to p_1, q \to q_1\}$, and $\{p \to p_2, q \to q_2\}$, where "$\to$" denotes "moves towards" and $\overline{pp_1}$, $\overline{pp_2}$, $\overline{qq_1}$ and $\overline{qq_2}$ are all perpendicular to $\overline{pq}$. Then τ_p^l is computed as the smallest time that any of these choices requires for $\overline{pq}$ to reach either e_1 or e_2. Formally,

$$\tau_p^l = \min\{\min\{\frac{|\overline{pe_1}|}{v_m}, \frac{|\overline{qe_1}|}{v_m}\}, \min\{\frac{|\overline{pe_2}|}{v_m}, \frac{|\overline{qe_2}|}{v_m}\}, \max\{\frac{|\overline{pp_1}|}{v_m}, \frac{|\overline{qq_1}|}{v_m}\}, \max\{\frac{|\overline{pp_2}|}{v_m}, \frac{|\overline{qq_2}|}{v_m}\}\}. \quad (1)$$

Note that p may have to cross o to get to p_1 (p_2) (cf. Figure 5(b)). In this case, p needs to first get to an endpoint e_1 (e_2). Therefore, we define $|\overline{pp_1}|$ ($|\overline{pp_2}|$) as the sum of $|\overline{pe_1}|$ ($|\overline{pe_2}|$) and $|\overline{e_1p_1}|$ ($|\overline{e_2p_2}|$). Similar definition applies to $|\overline{qq_1}|$ ($|\overline{qq_2}|$) if $\overline{qq_1}$ ($\overline{qq_2}$) intersects o.

The following theorem guarantees the correctness of Equation 1 and hence no false dismissal will be introduced by this invisible time period lower bound based pruning.

Theorem 1. *Given two points p and q and a line segment $\overline{e_1e_2}$ such that $\overline{pq}$ intersects $\overline{e_1e_2}$ at r ($r \neq e_1, e_2$), for p and q to move to two points p_1 and q_1 so that $\overline{p_1q_1}$ intersects $\overline{e_1e_2}$ at e_1 while $\max\{|\overline{pp_1}|, |\overline{qq_1}|\}$ is minimized, $\overline{pp_1}$ and $\overline{qq_1}$ should both be perpendicular to $\overline{pq}$ if neither $\overline{pp_1}$ nor $\overline{qq_1}$ intersects $\overline{e_1e_2}$, otherwise p_1 and q_1 should be two points such that $\overline{pp_1}$ and $\overline{qq_1}$ are overlapped by $\overline{pe_1}$ and $\overline{qe_1}$, respectively.*

Proof. First we prove that $\overline{pp_1}$ and $\overline{qq_1}$ should both be perpendicular to $\overline{pq}$ if doing so does not result in either $\overline{pp_1}$ or $\overline{qq_1}$ intersecting $\overline{e_1e_2}$. Figure 5(a) illustrates how p_1 and q_1 are located in this case. Now assume that there are two points p' and q' that p and q may move to so that $\overline{p'q'}$ intersects e_1. We prove $\max\{|\overline{pp_1}|, |\overline{qq_1}|\} \leq \max\{|\overline{pp'}|, |\overline{qq'}|\}$. We denote the line that overlaps $\overline{p'q'}$ as l_{e1}. If l_{e1} also overlaps $\overline{p_1q_1}$, then $\max\{|\overline{pp_1}|, |\overline{qq_1}|\} \leq \max\{|\overline{pp'}|, |\overline{qq'}|\}$ is guaranteed by the fact that $\overline{pp_1} \perp \overline{p_1q_1}$ and $\overline{qq_1} \perp \overline{p_1q_1}$, which means $|\overline{pp_1}|$ ($|\overline{qq_1}|$) must be the shortest distance between p (q) and a point on l_{e1}. Otherwise (l_{e1} does not overlap $\overline{p_1q_1}$), as shown in Figure 5(a), we let p_3 and q_3 be two points on l_{e1} such that $\overline{pp_3} \perp l_{e1}$ and $\overline{qq_3} \perp l_{e1}$.

Then by definition we have $\max\{|\overline{pp_3}|, |\overline{qq_3}|\} \le \max\{|\overline{pp'}|, |\overline{qq'}|\}$. We prove $\max\{|\overline{pp_1}|, |\overline{qq_1}|\} < \max\{|\overline{pp'}|, |\overline{qq'}|\}$ through proving $\max\{|\overline{pp_1}|, |\overline{qq_1}|\} < \max\{|\overline{pp_3}|, |\overline{qq_3}|\}$. The latter inequality is guaranteed by that p_3 and q_3 must be at different sides of $\overline{p_1q_1}$ because otherwise either $\overline{pp_3}$ or $\overline{qq_3}$ must cross $\overline{e_1e_2}$. Meanwhile, p and q are both at the same side of $\overline{p_1q_1}$. Therefore, either $\overline{pp_3}$ or $\overline{qq_3}$ must intersect $\overline{p_1q_1}$. Without lost of generality we assume that $\overline{pp_3}$ intersects $\overline{p_1q_1}$. Then in triangle $\triangle p_3p_1p$, we have $\angle p_3p_1p > 90°$ because $\angle q_1p_1p = 90°$. Thus, $|\overline{pp_3}|$ is the longest side in the triangle and $|\overline{pp_3}| > |\overline{pp_1}| = |\overline{qq_1}|$. Therefore, $\max\{|\overline{pp_1}|, |\overline{qq_1}|\} < \max\{|\overline{pp_3}|, |\overline{qq_3}|\}$.

Next we prove that if p and q moving along the lines that are perpendicular to the original $\overline{pq}$ will result in $\overline{pp_1}$ or $\overline{qq_1}$ intersecting $\overline{e_1e_2}$ before p and q are visible to each other, then p_1 and q_1 should change to two points such that $\overline{pp_1}$ and $\overline{qq_1}$ are overlapped by $\overline{pe_1}$ and $\overline{qe_1}$, respectively. This effectively means that p and q should move to e_1 directly. Once either p or q reaches e_1, the line that connects them will intersect $\overline{e_1e_2}$ at e_1. Therefore, we need to prove $\min\{|\overline{pe_1}|, |\overline{qe_1}|\} < \max\{|\overline{pp'}|, |\overline{qq'}|\}$. Without lost of generality we assume that $\overline{pp_1}$ intersects $\overline{e_1e_2}$ (note that $\overline{pp_1}$ and $\overline{qq_1}$ cannot both intersect $\overline{e_1e_2}$). Figure 5(b) illustrates the case, where l_{e1} is a line that overlaps $\overline{p'q'}$ and $\overline{pp_3}\perp l_{e1}, \overline{qq_3}\perp l_{e1}$ at p_3 and q_3, respectively. We prove $\min\{|\overline{pe_1}|, |\overline{qe_1}|\} < \max\{|\overline{pp'}|, |\overline{qq'}|\}$ through proving $|\overline{pe_1}| < \max\{|\overline{pp_3}|, |\overline{qq_3}|\}$. If $|\overline{pe_1}| < |\overline{pp_3}|$, then $|\overline{pe_1}| < \max\{|\overline{pp_3}|, |\overline{qq_3}|\}$ holds. Otherwise, we prove $|\overline{pe_1}| < |\overline{qq_3}|$. We draw a line segment $\overline{se_1}$ such that $\overline{se_1}\perp l_{e1}$ at e_1 and $\overline{se_1}$ interests $\overline{pq}$ at s. Then $|\overline{pe_1}| < |\overline{se_1}|$ holds because in $\triangle spe_1$, $\angle spe_1 > \angle spp_1 = 90°$. Meanwhile, in trapezoid e_1sqq_3, $|\overline{se_1}| < |\overline{qq_3}|$ because $\angle qse_1 > 90°$ (derived from $\angle spe_1 > 90°$ and $\angle pse_1 < 90°$). Thus, we have $|\overline{pe_1}| < |\overline{qq_3}|$ and therefore, $\min\{|\overline{pe_1}|, |\overline{qe_1}|\} < \max\{|\overline{pp'}|, |\overline{qq'}|\}$.

□

Until now we have considered only one obstacle between p and q. When there are multiple obstacles, we just need to compute a lower bound of τ_p for each of the obstacles using Equation 1, and then choose the smallest one as the overall lower bound τ^l_p. For example, in Figure 6, we first compute two invisible time period lower bounds for p based on $\overline{e_1e_2}$ and $\overline{e_3e_4}$, respectively. Then we can use the smaller one between the two lower bounds as the overall lower bound. The correctness of doing so is straightforward and hence the proof is omitted.

4.2.2. Moving Direction aware Invisible Time Period

In this subsection we improve the lower bound of the invisible time period by taking the query object's movement into consideration. The intuition of this lower bound is that usually a moving object will not change its moving direction dramatically and hence it will stay in the range of its current moving direction for a while. Within this range, we can compute a shortest path that p can reach the visible region and hence an invisible time period lower bound. This shortest path may not be the same as the overall shortest path as computed based on Theorem 1 in the last subsection, since the range p moving into may not enclose the overall shortest path. Thus, we can usually obtain a better lower bound of τ_p in this way. We call the resultant lower bound the moving direction aware invisible time period and denote it by τ^m_p.

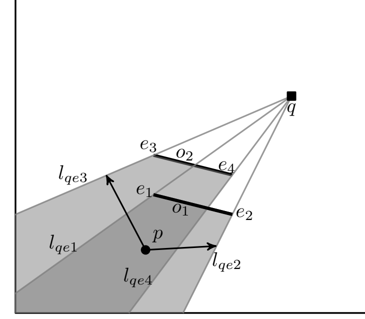


Figure 6: Computation of τ^l_p under multiple obstacles

Figure 7 shows an example. When a lower bound of τ_p is to be computed, we draw two lines l_{qe_1} and l_{qe_2} passing through the two endpoints e_1 and e_2 of an obstacle o, and intersecting each other at q. These two lines divide the movement of q into four ranges D_1, D_2, D_3 and D_4. Each of these ranges is like a safe region for the moving direction of q. As long as q remains in the range where it is moving into at this instant (D_1 as in Figure 7), we can compute a larger lower bound of τ_p based on the position relationship between this range, the obstacle o and the moving object p. This larger lower bound is the moving direction aware invisible time period τ^m_p. If q moves out of the range that it is currently moving into before τ^m_p expires, then we can simply fall back to the lower bound τ^l_p computed by the method discussed in Section 4.2.1.

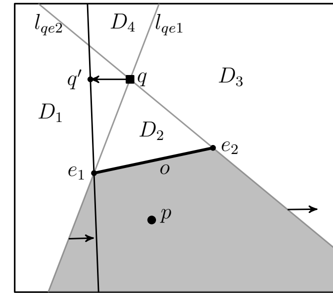


Figure 7: An example of the moving ranges

Next we illustrate how to compute τ^m_p. Again we consider how p and q should move so that $\overline{pq}$ can meet one of the endpoints e_1 and e_2 of the obstacle o as early as possible, but now the movement of q is constrained by the range it is moving into. Suppose q is moving into range D_1, as shown in Figure 7. We first consider how $\overline{pq}$ can meet e_1 early. Effectively this means how p can reach the left boundary of the invisible region of q, line l_{qe_1}. An observation is that, for any point in D_1, say q', q moving to q' will make l_{qe_1} rotate towards p. We need to find the optimal path for q that makes l_{qe_1} rotate the fastest to meet p, and the optimal path for p to reach l_{qe_1} accordingly. Meanwhile, q moving to q' will make the right boundary of the invisible region of q, line l_{qe_2}, rotate away from p. To let $\overline{pq}$ meet e_2 early, we need to find the optimal path for q that makes l_{qe_2} rotate the least and the optimal path for p to reach l_{qe_2} as early as possible.

Next we describe how these optimal paths are computed. Since l_{qe_1} and l_{qe_2} have different rotation directions for different moving ranges, we analyze the different cases of the four moving ranges separately and use Figure 8 for the illustration.

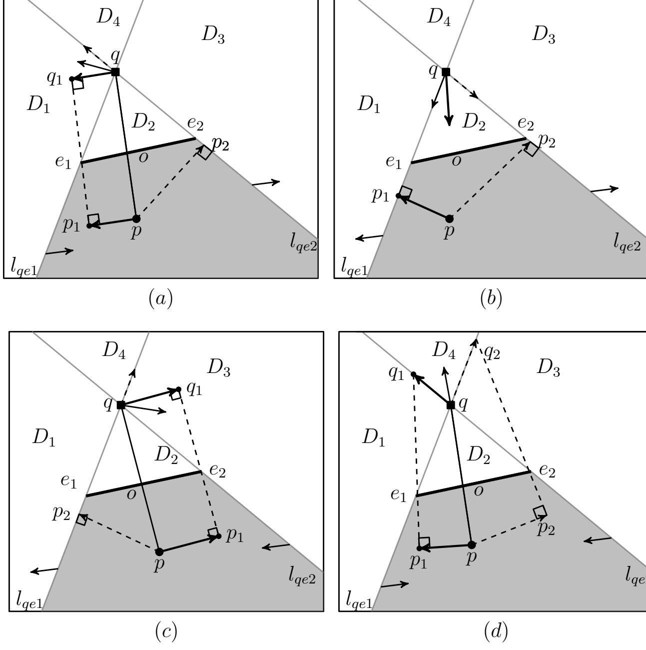


Figure 8: Computation of τ_p^m

As shown in Figure 8(a), if q is moving towards D_1, then l_{qe_1} will rotate towards p while l_{qe_2} will rotate away from p. We describe how p can reach l_{qe_1} and l_{qe_2} early as follows. (i) For p to reach l_{qe_1} early, p and q should both follow the movement as described in the last subsection, i.e., both p and q move in the direction that is perpendicular to the current $\overline{pq}$, and if in this way p or q reaches o before $\overline{pq}$ reaches e_1, then both p and q should move directly to e_1. (ii) For p to reach l_{qe_2} early, since q moving into D_1 will result in l_{qe_2} rotating away from p, the best q can do is to move along the current l_{qe_2} (or do not move at all) so that l_{qe_2} does not rotate away any farther. Meanwhile, p should move perpendicularly towards l_{qe_2} since this is the shortest path that a point can reach a line. If in this way p reaches o before $\overline{pq}$ reaches e_2, then p should move directly to e_2. The shortest time that p and q needs to move as described above before $\overline{pq}$ reaches either e_1 or e_2 (i.e., p reaches either l_{qe_1} or l_{qe_2}) defines τ_p^m.

If q is moving towards D_2, as shown in Figure 8(b), both l_{qe_1} and l_{qe_2} are rotating away from p. Therefore, for p to reach either line early, q should move along l_{qe_1} (l_{qe_2}) towards e_1 (e_2) to keep l_{qe_1} (l_{qe_2}) from rotating away, while p should move perpendicularly towards l_{qe_1} (l_{qe_2}). If in this way p reaches o before $\overline{pq}$ reaches e_1 (e_2), then p should move directly to e_1 (e_2).

If q is moving towards D_3, as shown in Figure 8(c), the situation is very similar to that of q moving towards D_1. Therefore, the analysis of that previous case (i.e., q moving towards D_1) applies. The only difference is that now l_{qe_1} is rotating away from p while l_{qe_2} is rotating towards p.

If q is moving towards D_4, as shown in Figure 8(d), both l_{qe_1} and l_{qe_2} are rotating towards p. In this case, q needs to move along the current l_{qe_2} (l_{qe_1}) so that l_{qe_1} (l_{qe_2}) can rotate the fastest towards p. To determine the shortest distance p and q need to move until $\overline{pq}$ intersects e_1, we let q_1 be a point on l_{qe_2} and $\overline{q_1p_1}$ be a line segment that intersects e_1 and is perpendicular to $\overline{pp_1}$. When $|\overline{qq_1}| = |\overline{pp_1}|$, the two line segments $\overline{qq_1}$ and $\overline{pp_1}$ are the shortest paths for $\overline{pq}$ to reach e_1. The correctness of this claim can be proved in a way that is similar to the proof of Theorem 1 and thus the proof is omitted. Intuitively, among all line segments in D_4 that have the same length as $\overline{qq_1}$, $\overline{qq_1}$ has the longest projection on the overall best path of q to make $\overline{pq}$ reach e_1 (i.e., $\overline{qq_1}$ in Figure 8(a)). Meanwhile, we have established in Theorem 1 that $\overline{pp_1}$ and $\overline{qq_1}$ should be of the same length so that $\max\{|\overline{pp_1}|, |\overline{qq_1}|\}$ is minimized. Therefore, the paths $\overline{pp_1}$ and $\overline{qq_1}$ described above are the optimal paths. By basic geometry we compute the locations of p_1 and q_1 and get $\overline{pp_1}$ and $\overline{qq_1}$. Again if p cannot reach p_1 without crossing o then p should move directly to e_1. Similarly we get the optimal paths $\overline{pp_2}$ and $\overline{qq_2}$ for $\overline{pq}$ to reach e_2 early.

The correctness of the above analysis is supported by Theorem 1. We still use Equation 1 to compute τ_p^m, but the points represented by p_1 and p_2 in the equation are changed to the points described above for the four cases accordingly.

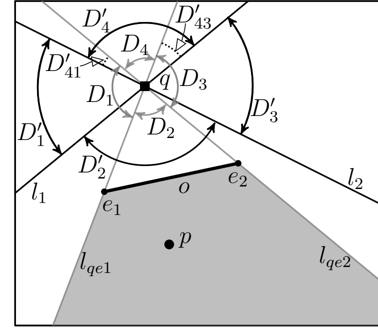


Figure 9: Another choice of data space partitioning

Discussion. Using l_{qe_1} and l_{qe_2} is not the only way to divide the space. We have chosen these two lines because every resultant range has one unchanged rotation direction for each of the invisible region sides. This leads to a simple and efficient way to compute a lower bound of τ_p. We might use other lines to partition the space and compute lower bounds of τ_p based on those lines, but then some of the resultant ranges may overlap more than one of the ranges as divided by l_{qe_1} and l_{qe_2}, and the computation of a lower bound of τ_p will become complicated and less efficient. For example, as shown in Figure 9, we use two lines l_1 and l_2 to divide the space and denote the resultant ranges by D'_1, D'_2, D'_3 and D'_4. We can see that D'_4 overlaps D_1, D_3 and D_4. The overlapping ranges are D'_{41}, D'_{43} and D_4, respectively. Then, we have to compute lower bounds of τ_p for each of these ranges and then find the smallest as the the lower bound of τ_p for range D'_4, which is more complex than using l_{qe_1} and l_{qe_2} directly.

5. Cost Analysis

In this section we analyze the performance of the proposed pruning methods and compare the proposed framework with the

snapshot VkNN based method in communication and computation costs. For simplicity, we denote the snapshot VkNN based method, the safe region based pruning method, the basic lower bound invisible time period based pruning method and the moving direction aware invisible time period based pruning method as **SV, SR, LITP** and **MITP**, respectively.

5.1. Communication Cost

In a client-server based query processing system, the communication cost is caused by two operations, i.e., soliciting location updates from the moving objects (clients) and reporting the query result to the query object. In a centralized system, the communication cost is replaced by the cost of accessing the object location data within the centralized system and reporting the query result to itself for later use. Since the cost of reporting the query result is the same for all CVkNN query processing methods, and it is much smaller than the cost of getting the location updates, we focus on the cost of getting the location updates. We compare for the various methods the number of location updates solicited during a period of T timestamps starting from the beginning of the period.

A) **SV** needs the exact location for each moving object (including the query object) at each timestamp. Therefore, its communication cost for T timestamps, denoted as CM_{SV}, is computed as:

$$CM_{SV} = (|\mathcal{P}| + 1)T = (|\mathcal{P}| + 1) + (|\mathcal{P}| + 1)(T - 1). \qquad (2)$$

B) **Our proposed framework** first performs safe region based pruning using the locations of all objects, which results in a subset S_c of $\mathcal{P}$ as the query answer candidate set for the next T timestamps (including the current timestamp). Then invisible time period based pruning is performed to further reduce the number of objects in S_c to be checked by the refinement step to generate the exact query answer set at each timestamp.

B.i) If only **SR** is applied, then the communication cost of the proposed framework, denoted by CM_{SR}, is computed as:

$$CM_{SR} = (|\mathcal{P}| + 1) + (|S_c| + 1)(T - 1) \qquad (3)$$

By comparing Equations 2 and 3, we can see that $CM_{SR} \leq CM_{SV}$ is guaranteed since $S_c \subseteq \mathcal{P}$. The superiority of the proposed framework over SV depends on $\frac{|S_c|}{|\mathcal{P}|}$, which in turn depends on T because T determines the sizes of the safe regions and the pruning distance. To learn the effect of T, we look at the average per timestamp communication cost of SR, denoted as $\overline{CM_{SR}}$ and derived from Equation 3 as follows.

$$\begin{aligned}\overline{CM_{SR}} &= \frac{(|\mathcal{P}| + 1) + (|S_c| + 1)(T - 1)}{T} \\ &= \frac{|\mathcal{P} \setminus S_c| + (|S_c| + 1) + (|S_c| + 1)(T - 1)}{T} \\ &= \frac{|\mathcal{P} \setminus S_c|}{T} + |S_c| + 1\end{aligned} \qquad (4)$$

When T increases, the denominator in the equation increases. However, $|S_c|$ also becomes larger because the size of the safe regions increases and so as the pruning distance. There is no obvious theoretical trend on the combined effect and we will use experiments to find a suitable value of T to optimize the performance of the proposed framework.

We let ω_{SR} be $\frac{|S_c|}{|\mathcal{P}|}$ and use it to denote the pruning power of SR. Then CM_{SR} becomes $(|\mathcal{P}|+1)+(\omega_{SR}|\mathcal{P}|+1)(T-1)$. When the invisible time period based pruning is also applied, the size of $|S_c|$ is further reduced.

B.ii) The communication cost of the proposed framework when applying **SR + LITP** and **SR + MITP**, denoted as CM_{SL} and CM_{SM} respectively, are computed as follows.

$$CM_{SL} = (|\mathcal{P}| + 1) + (\omega_{SL}|\mathcal{P}| + 1)(T - 1) \qquad (5)$$

$$CM_{SM} = (|\mathcal{P}| + 1) + (\omega_{SM}|\mathcal{P}| + 1)(T - 1) \qquad (6)$$

Here, ω_{SL} and ω_{SM} denote the pruning power of SR in combined with LITP and MITP, respectively. The pruning power depends on whether LITP or MITP can keep more objects from S_c. By definition, MITP computes invisible time period lower bounds that are at least as long as what LITP computes. Thus, its pruning power is at least as good as that of LITP, i.e., $\omega_{SM} \leq \omega_{SL}$. Therefore, SR+MITP performs no worse than SR+LITP in terms of communication cost.

5.2. Computation Cost

For computation cost analysis we also consider the cost of a period of T timestamps.

A) **SV** computes the visible distance and performs a sorting for all objects at each timestamp. The computation cost, denoted as CP_{SV}, is computed as:

$$CP_{SV} = \varphi_{SV}|\mathcal{P}|T = \varphi_{SV}|\mathcal{P}| + \varphi_{SV}|\mathcal{P}|(T - 1), \qquad (7)$$

where φ_{SV} denotes the scaling of the cost of visible distance computation and sorting on $|\mathcal{P}|$.

B) **Our proposed framework**'s computation cost consists of three types of costs: (i) the cost of safe region based pruning, denoted as CP_{SR}, which involves building an in-memory R-tree and a best-first traversal on the tree, (ii) the cost of invisible time based pruning, denoted as CP_{LP} and CP_{MP} for LITP and MITP, respectively, which involves invisible time period computation and checking whether the objects in S_c are in their invisible time periods, and (iii) the cost of refinement, denoted as CP_{RF}, which involves computing the exact distance between the query object and the objects in S_c and sorting to determine the query answer set. While CP_{SR} is required only at the first timestamp, CP_{LP} (CP_{MP}) and CP_{RF} are both required at every timestamp. We denote the computation cost for SR + LITP and SR + MITP as CP_{SL} and CP_{SM}, respectively. Then we have:

$$\begin{aligned}CP_{SL} &= CP_{SR} + (CP_{LP} + CP_{RF})T \\ &= CP_{SR} + CP_{LP} + CP_{RF} + (CP_{LP} + CP_{RF})(T - 1)\end{aligned} \qquad (8)$$

$$\begin{aligned}CP_{SM} &= CP_{SR} + (CP_{MP} + CP_{RF})T \\ &= CP_{SR} + CP_{MP} + CP_{RF} + (CP_{MP} + CP_{RF})(T - 1)\end{aligned} \qquad (9)$$

We first compare **SR + LITP** and **SR + MITP** based on the two equations and then compare them with SV. For SR + LITP and SR + MITP, the main difference is the strategy used for invisible time period lower bound computation. Specifically, MITP has a higher cost to compute an invisible time period lower bound because it has to considered a more complex case as shown in Figure 8(d). However, as discussed in the last subsection, MITP computes longer invisible time period lower bounds and hence needs to be invoked for a smaller number of times. Meanwhile, longer invisible time period lower bounds means smaller query answer candidate sets to be checked by the refinement step. Therefore, CP_{RF} is smaller for SR + MITP. All factors combined, SR + MITP is expected to outperform SR + LITP in most cases, as verified by the experiments.

We now compare **our framework** with SV. The computation cost of our our framework at the first timestamp during T, $CP_{SR} + CP_{LP}(CP_{MP}) + CP_{RF}$, is comparable to the average per timestamp cost of SV in the sense that both methods require some computation on the whole data set $\mathcal{P}$, and the time complexities are both at $O(|\mathcal{P}| \log |\mathcal{P}|)$ (R-tree building v.s. sorting). The speedup in pruning achieved from the R-tree built by our framework compensates the cost of building the tree. Meanwhile, in the following (T-1) timestamps, our filtering and refinement are based on a much smaller data set S_c compared with $\mathcal{P}$ for SV. Therefore, the advantage of our framework is explicit. As shown in the experimental study, our framework constantly outperforms SV by an order of magnitude.

6. Experiments

In this section we study the empirical performance of the proposed framework. We first describe the experimental settings in Section 6.1. Then we evaluate the impact of T and choose a suitable value of T for the framework in Section 6.2. We investigate the performance of the framework under various settings in Sections 6.3. In Section 6.4, we evaluate the cost of maintaining the in-memory R-tree required by the framework.

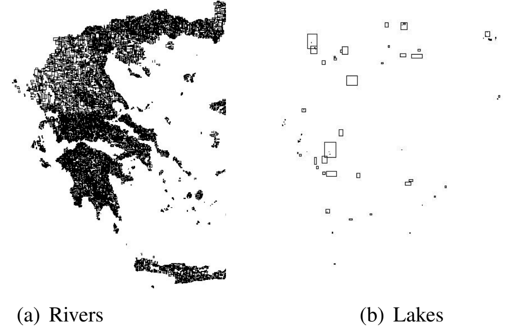
6.1. Experimental Settings

Parameter	Values
Data domain	**[0, 20000]** × **[0, 20000]**
T	1, 2, **4**, 8, 16, 32, 64
k	1, 5, 10, 50, **100**
v_m	2.5, 5, **10**, 25, 50
$\lvert\mathcal{P}\rvert$	100, 500, 1000, 5000, **10000**
$\lvert O\rvert$	Rivers: 1000, 5000, 10000, **24650**; Lakes: 10, 30, 50, **77**

Table 2: Parameters and their values

All algorithms were implemented in C++, and the experiments were conducted on a desktop computer with an Intel 2.4GHz CPU and 2GB memory.

We use two real data sets from the R-tree Portal[2] that contains bounding rectangles of 24,650 rivers and 77 lakes in Greece as



(a) Rivers (b) Lakes

Figure 10: Obstacle sets

our obstacle sets (cf. Figure 10). We call them the Rivers and the Lakes, respectively. We take the diagonals of the rectangles as the obstacles and map them into a domain of size [0, 20,000] × [0, 20,000].

We generate objects that move around the obstacles. Two types of moving object sets are generated, the uniform data sets and the Zipfian data sets, where the objects initially follow uniform and Zipian distributions, respectively. We use "RU" to denote experiments where Rivers is used as the set of obstacles and a uniform data set is used as the set of moving objects. Similarly, we use "RZ", "LU" and "LZ" to denote experiments done on Rivers and a Zipfian data set, Lakes and a uniform data set, and Lakes and a Zipfian data set, respectively. The data objects move randomly in the data domain with a global maximum speed that ranges from 2.5 to 50. To evaluate the effect of data set size, we vary the moving object set size from 100 to 10,000 and vary the obstacle set size from 10 to 24,650 (by sampling from the real data sets). We vary the value of k from 1 to 100 and the value of T from 1 to 64 to evaluate the impact of these two parameters. Table 2 summarizes the parameters used, where values in bold denote the default values.

We evaluate the performance of four different methods:

- SV, the straightforward snapshot VkNN based method.
- SR, the query processing framework with only the safe region based pruning enabled.
- SR+LITP, the query processing framework with the safe region based pruning and the basic lower bound invisible time period based pruning enabled.
- SR+MITP, the query processing framework with the safe region based pruning and the moving direction aware invisible time period based pruning enabled.

In these methods, when an R-tree is required, we use the R*-tree [5] implementation.

In all experiments, we run the different algorithms for a CVkNN query over a period of 300 timestamps. We measure the communication cost by counting the number of object location updates solicited by the query processor and the computation cost by recording the query processing time.

6.2. Choosing the Value of T

We first study the effect of T. We omit SV in this subsection as its performance is not related to T. As shown in Figure 11,

[2] http://www.chorochronos.org/

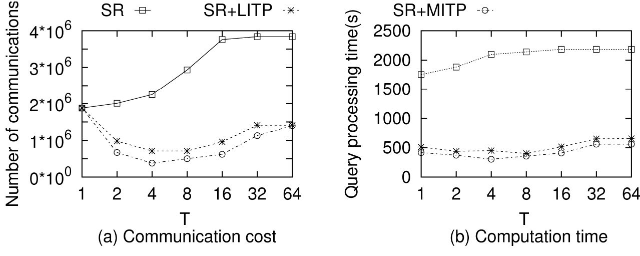


Figure 11: Effect of T

while the communication and computation costs of SR first increase and then become stable with the increase in the value of T, those of SR + LITP and SR + MITP first drop until T reaches 4 and then start to increase. This phenomena confirms the cost analysis in that the effect of T is twofold and there is no unified trend of the query performance when the value of T is varied. Since at $T = 4$ the three methods show better performance in general, we use it in the experiments in the following subsections.

We also notice that in the figure, compared with the performance difference in query processing time, the performance difference in the number of communications is relatively smaller for the three methods. This is because the number of communications scales linearly with the size of the query answer candidates $|S_c|$ while the query processing time scales in $O(|S_c| \log |S_c|)$ for the visible distance based sorting of the objects in S_c. Besides, when T is small, the operation of soliciting location updates for all objects is performed more frequently for each method and it dominates the communication cost. The difference in the communication cost of different methods becomes smaller. In the extreme case where $T = 1$, all methods have the same communication cost since they all solicit location updates from all objects at every timestamp.

6.3. Comparing Different Methods

In this subsection we compare the performance of the studied methods in different settings by varying the experiment parameters. When a parameter is varied, the other parameters stay with their default values.

6.3.1. Varying the Number of Moving Objects

We first evaluate the query processing performance when the number of moving objects ($|\mathcal{P}|$) is varied. As shown in Figure 12 and Figure 13, the number of communications and the query processing time increase with the increase of $|\mathcal{P}|$ for all methods. We observe that the proposed framework outperforms SV constantly, and when SR and MITP are applied, the advantage is the most significant, i.e., by an order of magnitude. An important observation is that, when the invisible time period based pruning techniques are used (i.e., SR+LITP and SR+MITP), the increase in the query processing costs is very slow when $|\mathcal{P}|$ increases. When $|\mathcal{P}|$ reaches 10,000, the query processing time of SR+LITP and SR+MITP stays within tens of seconds (i.e., below 0.1 seconds for processing the query at each timestamp on average) while that of SV is at hundreds or even thousands of seconds (i.e., over 1 second for processing the query at each timestamp on average). We also observe that,

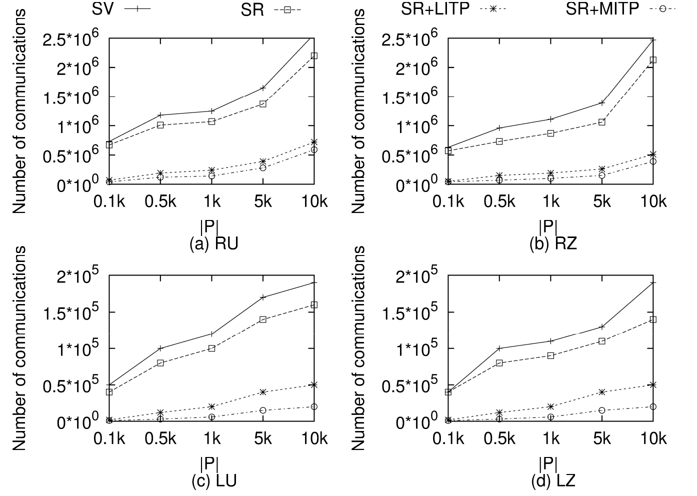


Figure 12: Number of communications v.s. $|\mathcal{P}|$

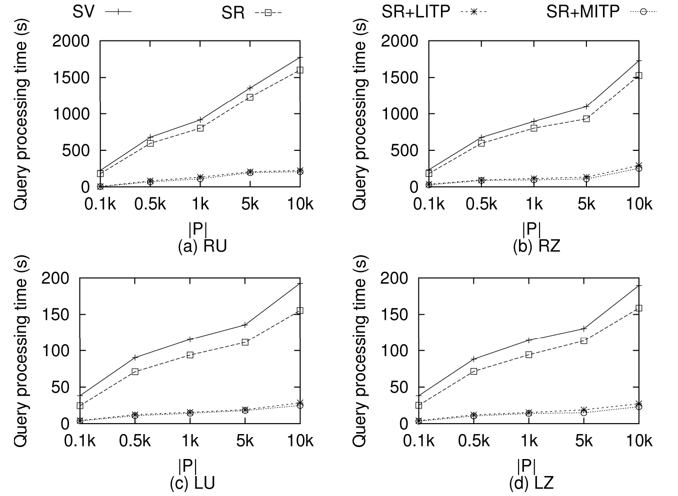


Figure 13: Query processing time v.s. $|\mathcal{P}|$

SR+MITP outperforms SR+LITP in terms of both communication and computation costs for most of the experiments, which validates the proposal of MITP and confirms the cost analysis in Section 5.

6.3.2. Varying the Number of Obstacles

Next we vary the number of obstacles ($|O|$) by randomly sampling obstacles from the Rivers and Lakes. Since Lakes contains a much smaller number of obstacles than Rivers does, we use smaller sets of moving objects (i.e., $|\mathcal{P}| = 1000$) in the "LU" and "LZ" experiments. Again, as shown in Figure 14 and Figure 15, the proposed framework outperforms SV for all cases and SR+MITP shows the best performance for most cases. An observation is that when $|O|$ increases the query costs of different methods vary in different patterns. Specifically, the communication cost of SV stays the same when $|O|$ is varied because SV always solicit location updates from all objects. Meanwhile, the communication costs of the other three methods first increase and then decrease. This is because when $|O|$ increases, it may bring a larger $|S_c|$ because a larger $|O|$ means more difficult to find k safe regions that are entirely visible to the query ob-

ject, which may result in a larger PD and hence a larger $|S_c|$. It may also bring a smaller $|S_c|$ because most of the safe regions of the objects may simply be blocked from the query object by the obstacles. The latter effect is more explicit when $|O|$ is large enough for the obstacles in O to cover a large portion of the space. As for the query processing time, the increase in $|O|$ results in the increase of the query processing time for all methods because the cost of checking object visibility increases.

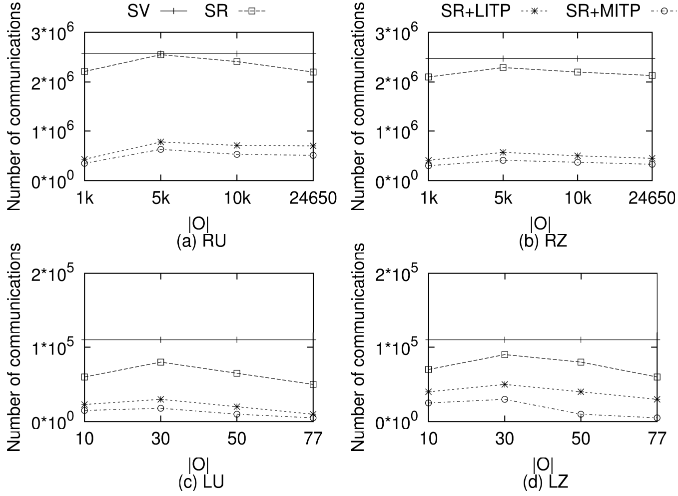


Figure 14: Number of communications v.s. $|O|$

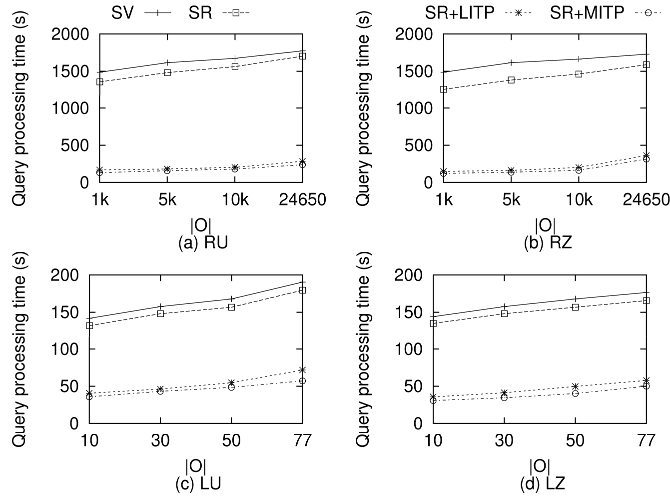


Figure 15: Query processing time v.s. $|O|$

6.3.3. Varying the Global Maximum Speed

We show the effect of the global maximum speed of the moving objects (v_m) on the query processing costs in Figure 16 and Figure 17. We observe that the increase in v_m results in the increase in the query processing costs for the proposed framework. This is expected because the increase in v_m results in the increase in the sizes of the safe regions as well as decrease in the lengths of the invisible time periods. The pruning power of SR, LITP and MITP suffers and the query processing efficiency is lowered. However, even when $v_m = 50$, which is quite large considering the size of the data domain, the proposed framework still outperforms SV significantly, which again confirms the superiority of the framework.

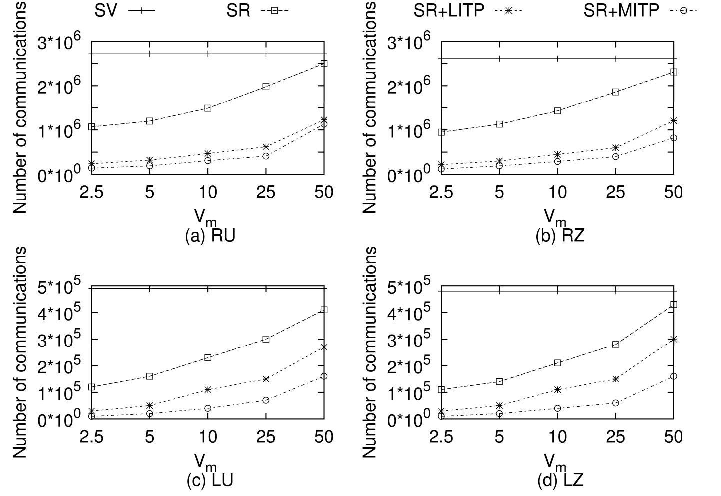


Figure 16: Number of communications v.s. v_m

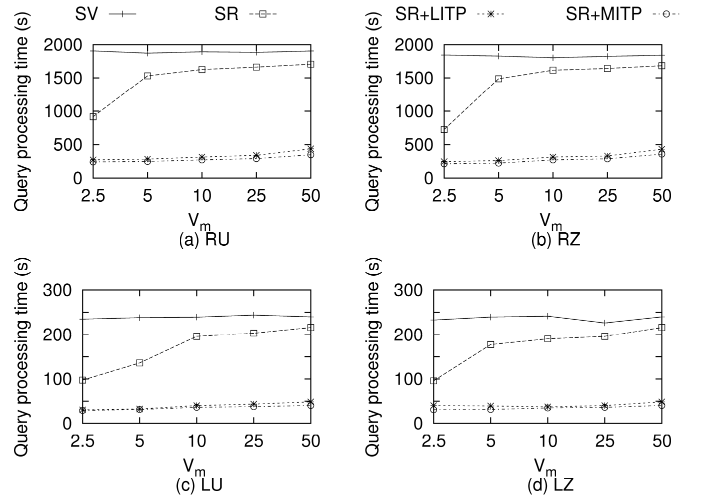


Figure 17: Query processing time v.s. v_m

6.3.4. Varying Query Parameter k

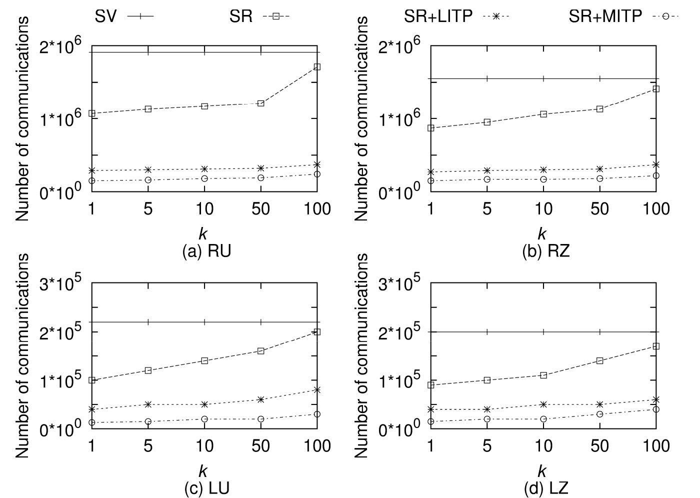


Figure 18: Number of communications v.s. k

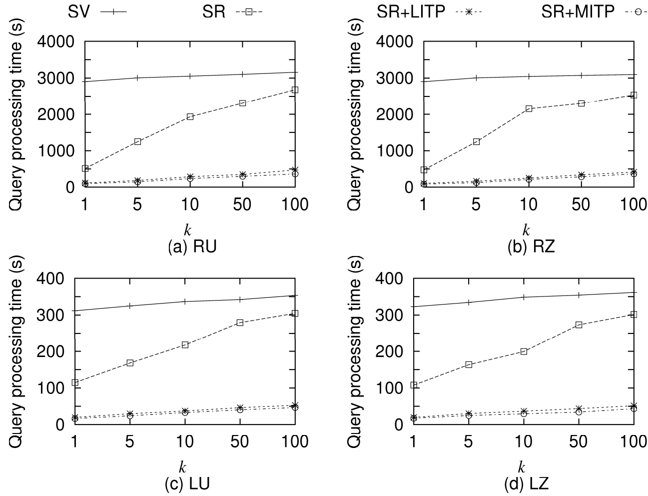


Figure 19: Query processing time v.s. k

Figure 18 and Figure 19 show the query processing costs for the four methods when the query parameter k is varied. As can be seen from the figures, the proposed framework outperforms SV in terms of both communication and computation costs when k varies from 1 to 100. An important advantage of the proposed framework is that when both safe region based pruning and invisible time period based pruning are applied (i.e., SR+LITP and SR+MITP), the query processing costs stay relatively stable when k increases. This demonstrates the scalability of the proposed framework.

6.3.5. Varying the Object Moving Pattern

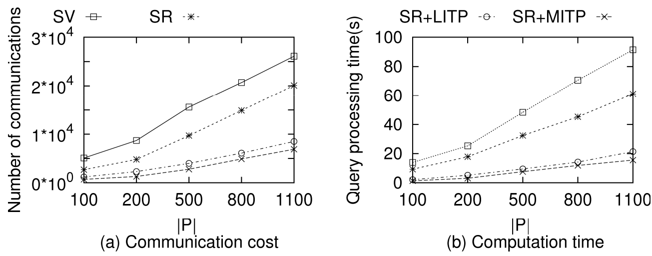


Figure 20: Algorithm performance on a real dataset

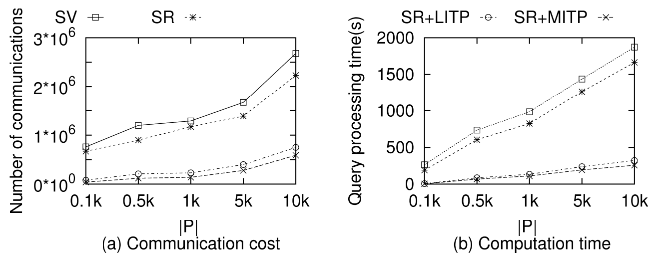


Figure 21: Algorithm performance in a small data space

In this subsection we vary the object moving pattern by (i) using a real trajectory dataset to model the object movements and (ii) changing the data space size.

Experiments on a real data set. The real trajectory data set used is the *Trucks* data set from the R-tree Portal. It contains 1,100 trajectories of trucks delivering concrete to several construction sites around Athens metropolitan area in Greece. We map the trajectories to a $[0, 20, 000] \times [0, 20, 000]$ domain, and use each trajectory to model the movement of a data object. We randomly pick a trajectory for the query object. We use the Spatial Data Generator[3] from the R-tree Portal to generate 500 obstacles randomly placed in the data space where there is no trajectory passing. The size of each obstacle is randomly chosen from the range of $[500, 1, 000]$. Figure 20 shows the experimental result where we vary the number of moving objects from 100 to 1,100. We can see that our proposed algorithms (SR, SR+LITP, and SR+MITP) outperform the baseline algorithm SV constantly, and the advantage is more significant as more pruning techniques are applied.

Experiments in a small data space. We perform experiments in a small geographical area where the moving object density is high. This is to evaluate our algorithm performance under our motivating application scenario, the MMOFPS game scenario. Due to the limited availability of real MMOFPS game trajectory data, we use the Spatial Data Generator to generate objects and obstacles in a $[0, 500] \times [0, 500]$ domain. We generate 10,000 obstacles randomly placed in the data space. The size of each obstacle is randomly chosen in the range of $[5, 10]$. We vary the number of data objects from 100 to 10,000. The objects (including the query object) are randomly placed in the data space initially, and then move towards a random direction with a speed randomly chosen in the range of $[0, 5]$. The velocity of an object is reinitialized when the object hits an obstacle. Figure 21 shows the experimental result. We can see that our proposed algorithms again outperform the baseline algorithm. This result confirms the robustness of our algorithms in small geographical areas with high object density.

We also varied other experimental settings (e.g., the number of obstacles) on the real data set and in the small data space. The results show similar patterns and hence are omitted.

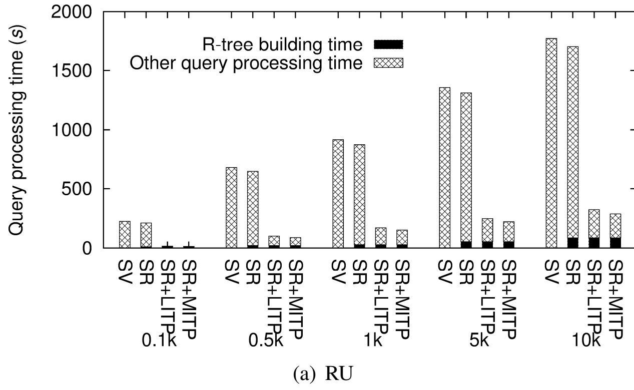
6.4. The Cost of Maintaining an In-memory R-tree

In this subsection we evaluate the cost of maintaining an in-memory R-tree by the proposed framework on data sets of different sizes.

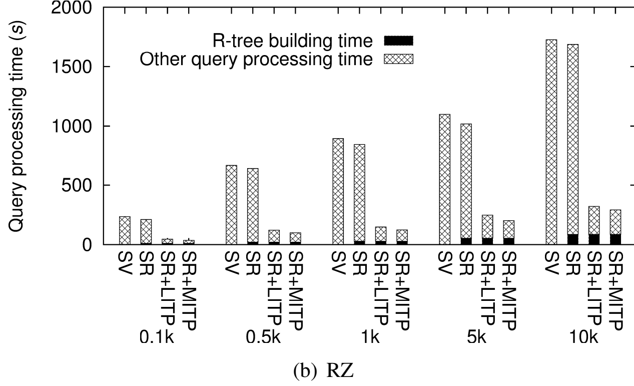
Rebuilding the R-tree. We first report the time used for building an R-tree compared with the overall query processing time. As shown in Figure 22, the time required for building an R-tree is very small in general. Even for a data set size as large as 10,000, building an in-memory R-tree only takes 0.33 seconds (during the 300 timestamps that we ran the CVkNN query with a period length T of 4 timestamps, the R-tree is built 75 times and it takes 25 seconds in total as shown in the figure). Considering that even with this overhead, SR+LITP and SR+MITP still have much smaller query processing time than that of SV, building an R-tree once every T timestamps is justified.

Maintaining the R-tree incrementally. We then compare the query processing performance of rebuilding the R-tree, denoted by "SR+MITP-R", with that of maintaining the R-tree

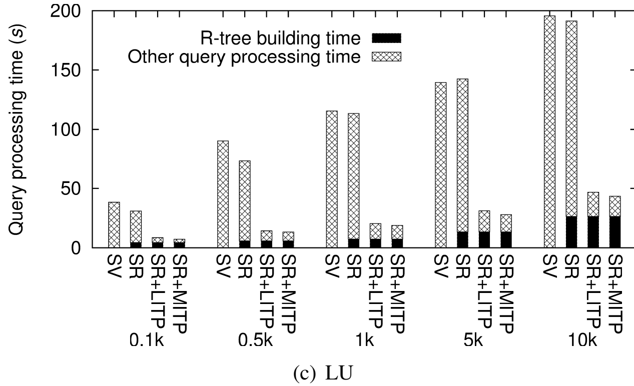
[3]http://www.chorochronos.org/?q=node/49



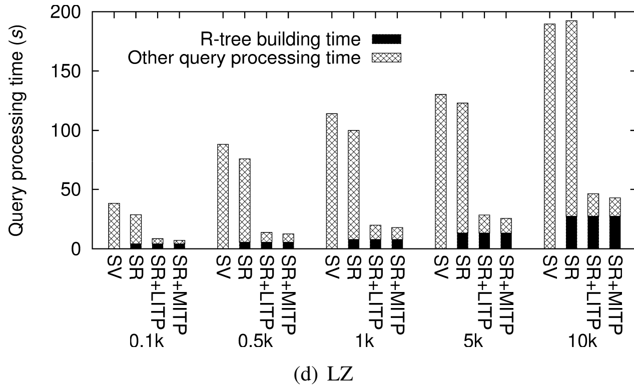
(a) RU



(b) RZ



(c) LU



(d) LZ

Figure 22: Cost of building an R-tree

incrementally, denoted by "SR+MITP-I". We add another baseline algorithm using the incremental computation technique proposed by Xiong *et al.* [31], denoted by SEA-CNN. SEA-CNN processes the CVkNN query as follows. It computes an

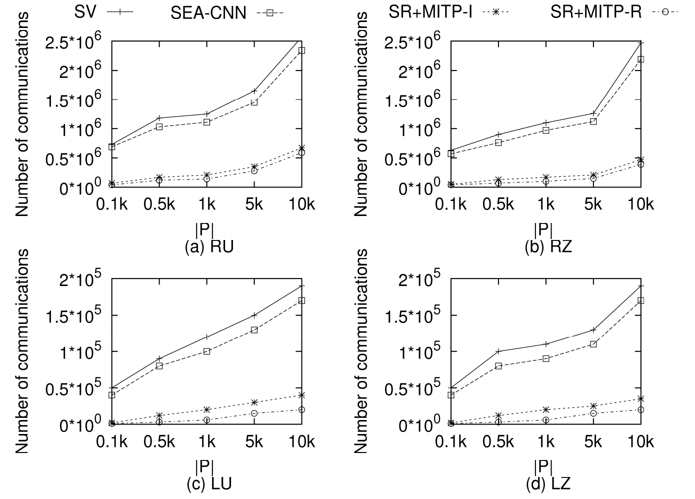


Figure 23: Number of communications v.s. update processing schemes (varying $|\mathcal{P}|$)

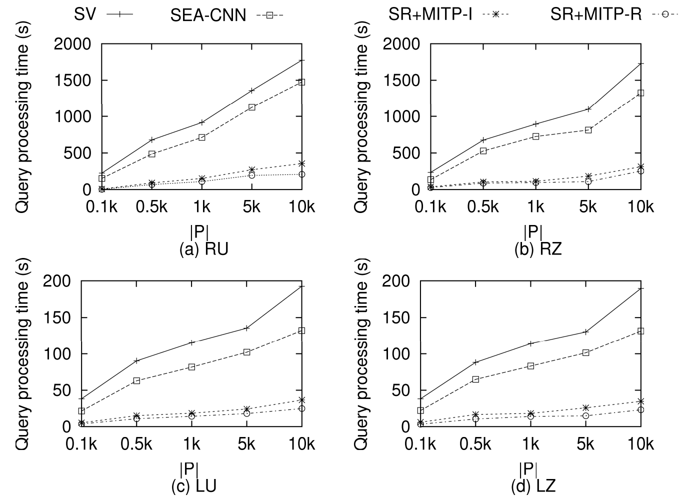


Figure 24: Query processing time v.s. update processing schemes (varying $|\mathcal{P}|$)

initial query answer at the start of the query. Then it processes the location updates of the objects one at a time and computes the effect of each individual update on the query answer. For example, if an object that is close to the query point but invisible moves and becomes visible, this object is added to the query answer to replace the existing farthest VkNN. If a current VkNN becomes invisible, a snapshot VkNN query is performed to update the query answer.

As shown in Figures 23 and 24, SR+MITP-R outperforms the two incremental update processing methods constantly in terms of both communication cost and query processing time. The advantage grows as the number of data objects increases. We observe that the performance of SEA-CNN is quite close to that of SV. This is because SEA-CNN requires a snapshot VkNN query to process an update that causes an existing VkNN to be removed from the current VkNN set. At each timestamp, there are many object updates. Thus, almost every timestamp requires a snapshot VkNN query and hence, SEA-CNN cannot perform much better than SV. SR+MITP-I also pro-

cesses every update, but it simply reinserts the updated object into the R-tree, which is much faster than a snapshot VkNN query. Therefore, SR+MITP-I is much faster than SEA-CNN. However, SR+MITP-I is still slower than SR+MITP-R because SR+MITP-R batch processes all updates on the R-tree during a T-timestamp period at once, while SR+MITP-I processes the updates one after another. When an object updates multiple times in a period, SR+MITP-I has to process each update to maintain the R-tree, while SR+MITP-R just needs one rebuild of the R-tree to reflect the impact of the multiple updates. Experiments where we vary other experimental settings (e.g., the maximum speed v_m) show similar results and hence are omitted.

7. Conclusions and Future Work

In this paper, we proposed a filtering-and-refinement framework to process the continuous visible k nearest neighbor query on moving objects. This framework utilizes spatial proximity and visibility properties between the moving objects to prune the query search space. Spatial proximity based pruning uses the safe region to compute a set of query answer candidates S_c that will be valid for a period of T timestamps. During this period, the query processor will only require location updates from the objects in S_c to compute the exact query answer. As a result, the communication cost for soliciting object location updates as well as the computation cost for examining the distance between the objects are reduced. Visibility based pruning then uses the invisible time period to reduce the number of objects in S_c that need to be examined at each timestamp so as to further reduce the query processing costs. As our cost analysis and experimental study show, the proposed query processing framework constantly outperforms a snapshot VkNN based query processing algorithm by an order of magnitude.

For future work we will consider extending our techniques to other types of obstacles (e.g., polygons) and distance metrics (e.g., Network distance).

References

[1] Alamri, S., Taniar, D., Safar, M.. Indexing moving objects in indoor cellular space. In: 15th International Conference on Network-Based Information Systems (NBiS). 2012. p. 38–44.

[2] Alamri, S., Taniar, D., Safar, M.. Indexing moving objects for directions and velocities queries. Information Systems Frontiers 2013;15(2):235–248.

[3] Alamri, S., Taniar, D., Safar, M.. Indexing of spatiotemporal objects in indoor environments. In: 27th International Conference on Advanced Information Networking and Applications (AINA). 2013. p. 453–460.

[4] Asano, T., Asano, T., Guibas, L., Hershberger, J., Imai, H.. Visibility-polygon search and euclidean shortest paths. In: 26th Annual Symposium on Foundations of Computer Science (SFCS). 1985. p. 155–164.

[5] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.. The r*-tree: an efficient and robust access method for points and rectangles. In: SIGMOD. 1990. p. 322–331.

[6] Benetis, R., Jensen, S., Karčiauskas, G., čaltenis, S.. Nearest and reverse nearest neighbor queries for moving objects. The VLDB Journal 2006;15(3):229–249.

[7] Biveinis, L., Šaltenis, S., Jensen, C.S.. Main-memory operation buffering for efficient r-tree update. In: VLDB. 2007. p. 591–602.

[8] Cheema, M.A., Lin, X., Zhang, Y., Wang, W., Zhang, W.. Lazy updates: an efficient technique to continuously monitoring reverse knn. PVLDB 2009;2(1):1138–1149.

[9] http://www.fpsreport.com/news/65-cross fire/, . Crossfire china breaks again record: 3.5 million. 2012.

[10] Gao, Y., Zheng, B., Chen, G., Lee, W.C., Lee, K.C.K., Li, Q.. Visible reverse k-nearest neighbor query processing in spatial databases. TKDE 2009;21(9):1314–1327.

[11] Gao, Y., Zheng, B., Chen, G., Li, Q., Guo, X.. Continuous visible nearest neighbor query processing in spatial databases. The VLDB Journal 2011;20(3):371–396.

[12] Gao, Y., Zheng, B., Lee, W.C., Chen, G.. Continuous visible nearest neighbor queries. In: EDBT. 2009. p. 144–155.

[13] Hashem, T., Kulik, L., Zhang, R.. Countering overlapping rectangle privacy attack for moving knn queries. Information Systems 2013;38(3):430–453.

[14] Hsueh, Y.L., Zimmermann, R., Wang, H., Ku, W.S.. Partition-based lazy updates for continuous queries over moving objects. In: GIS. 2007. p. 1–8.

[15] Hu, H., Xu, J., Lee, D.L.. A generic framework for monitoring continuous spatial queries over moving objects. In: SIGMOD. 2005. p. 479–490.

[16] Mokbel, M.F., Aref, W.G.. Sole: scalable on-line execution of continuous queries on spatio-temporal data streams. The VLDB Journal 2008;17(5):971–995.

[17] Mokbel, M.F., Xiong, X., Aref, W.G.. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In: SIGMOD. 2004. p. 623–634.

[18] Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.. A threshold-based algorithm for continuous monitoring of k nearest neighbors. TKDE 2005;17:1451–1464.

[19] Nutanong, S., Tanin, E., Zhang, R.. visible nearest neighbor queries. In: DASFAA. 2007. p. 876–883.

[20] Nutanong, S., Tanin, E., Zhang, R.. Incremental evaluation of visible nearest neighbor queries. TKDE 2010;22(5):665–681.

[21] Nutanong, S., Zhang, R., Tanin, E., Kulik, L.. The v*-diagram: a query-dependent approach to moving knn queries (2008). In: PVLDB. 2008. p. 1095–1106.

[22] Nutanong, S., Zhang, R., Tanin, E., Kulik, L.. Analysis and evaluation of v*-knn: an efficient algorithm for moving knn queries. The VLDB Journal 2010;19(3):307–332.

[23] Sharifzadeh, M., Shahabi, C.. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. PVLDB 2010;3(1):1231–1242.

[24] Song, Z., Roussopoulos, N.. K-nearest neighbor search for moving query point. In: SSTD. 2001. p. 79–96.

[25] Tao, Y., Papadias, D.. Time-parameterized queries in spatio-temporal databases. In: SIGMOD. 2002. p. 334–345.

[26] Tao, Y., Papadias, D., Shen, Q.. Continuous nearest neighbor search. In: VLDB. 2002. p. 287–298.

[27] Vatti, B.R.. A generic solution to polygon clipping. Communications of ACM 1992;35(7):56–63.

[28] Šaltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.. Indexing the positions of continuously moving objects. In: SIGMOD. 2000. p. 331–342.

[29] Šidlauskas, D., Šaltenis, S., Christiansen, C.W., Johansen, J.M., Šaulys, D.. Trees or grids?: indexing moving objects in main memory. In: SIGSPATIAL. 2009. p. 236–245.

[30] Xia, T., Zhang, D.. Continuous reverse nearest neighbor monitoring. In: ICDE. 2006. p. 77–.

[31] Xiong, X., Mokbel, M.F., Aref, W.G.. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE. 2005. p. 643–654.

[32] Yu, X., Pu, K.Q., Koudas, N.. Monitoring k-nearest neighbor queries over moving objects. In: ICDE. 2005. p. 631–642.

[33] Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.. Location-based spatial queries. In: SIGMOD. 2003. p. 443–454.

[34] Zhang, R., Jagadish, H.V., Dai, B.T., Ramamohanarao, K.. Optimized algorithms for predictive range and knn queries on moving objects. Information Systems 2010;35(8):911–932.

[35] Zhang, R., Qi, J., Lin, D., Wang, W., Wong, R.C.W.. A highly optimized algorithm for continuous intersection join queries over moving objects. The VLDB Journal 2012;21(4):561–586.