6-2011

Continuous Visible Nearest Neighbor Query Processing in Spatial Databases

Yunjun GAO
Zhejiang University

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

Gencai CHEN
Zhejiang University

Qing LI
City University of Hong Kong

Xiaofa GUO
Zhejiang University

DOI: https://doi.org/10.1007/s00778-010-0200-z

The VLDB Journal manuscript No.
(will be inserted by the editor)

Continuous visible nearest neighbor query processing in spatial databases

Yunjun Gao · Baihua Zheng · Gencai Chen · Qing Li · Xiaofa Guo

Received: date / Accepted: date

Abstract In this paper, we identify and solve a new type of spatial queries, called *continuous visible nearest neighbor* (CVNN) search. Given a data set P, an obstacle set O, and a query line segment q in a two-dimensional space, a CVNN query returns a set of $\langle p, R\rangle$ tuples such that $p \in P$ is the nearest neighbor to every point r along the interval $R \subseteq q$ as well as p is *visible* to r. Note that p may be NULL, meaning that all points in P are *invisible* to all points in R due to the obstruction of some obstacles in O. In contrast to existing continuous nearest neighbor query, CVNN retrieval considers the impact of obstacles on *visibility* between objects, which is ignored by most of spatial queries. We formulate the problem, analyze its unique characteristics, and develop efficient algorithms for *exact* CVNN query processing. Our methods (i) utilize conventional data-partitioning indices (e.g., R-trees, etc.) on both P and O, (ii) tackle the CVNN search by performing a *single* query for the *entire* query line segment, and (iii) only access the data points and obstacles *relevant to the final query result* by employing a suite of effective pruning heuristics. In addition, several interesting variations of CVNN queries have been introduced and they can be supported by our techniques, which further demonstrates the flexibility of the proposed algorithms. A comprehensive experimental evaluation using both real and synthetic datasets has been conducted to verify the effectiveness of our proposed pruning heuristics, and the performance of our proposed algorithms.

Keywords Query processing · Nearest neighbor · Visible · Spatial database · Algorithm

Yunjun Gao (Corresponding author) · Gencai Chen · Xiaofa Guo
College of Computer Science
Zhejiang University, Hangzhou, China
E-mail: {gaoyj, chengc, guoxf}@zju.edu.cn

Baihua Zheng
School of Information Systems
Singapore Management University, Singapore, Singapore
E-mail: bhzheng@smu.edu.sg

Qing Li
Department of Computer Science
City University of Hong Kong, Hong Kong, China
E-mail: itqli@cityu.edu.hk

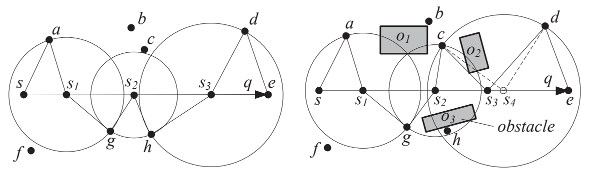
1 Introduction

The *continuous nearest neighbor* (CNN) search, an important operator in spatial databases, has been well-studied [1–3]. Given a set of points P and a query line segment q, a CNN query retrieves the nearest neighbor (NN) of every point on q. The result of CNN retrieval, denoted by $CNN(q)$, contains a set of $\langle p, R\rangle$ tuples, such that $p \in P$ is the NN of each point r along the interval $R \subseteq q$, i.e., $\forall r \in R$, $\forall p' \in P - \{p\}$, $dist(p, r) \leq dist(p', r)$[1]. An example is shown in Figure 1(a), where data set $P = \{a, b, c, d, f, g, h\}$ and query line segment $q = [s, e]$. $CNN(q) = \{\langle a, [s, s_1]\rangle$, $\langle g, [s_1, s_2]\rangle$, $\langle h, [s_2, s_3]\rangle$, $\langle d, [s_3, e]\rangle\}$, indicating that point a is the NN for any point along the interval $[s, s_1]$, point g is the NN for any point along the interval $[s_1, s_2]$, and so on. Points s_1, s_2, s_3 on q are called *split points*, as the NN object changes at those points.

Conventional CNN search does not take obstacles into consideration. However, many physical obstacles (e.g., buildings, blindages, and hills, etc.) exist in the real world, and their existence may affect the visibility/distance between objects and hence the result of spatial queries such as range query, NN search, and spatial join, etc. Furthermore, in some applications, users might be only interested in the objects that are *visible* or *reachable* to them.

Recently, the impact of obstacles has been studied in various spatial queries. Example queries include (i) *visible*

[1] Without loss of generality, $dist(p_i, p_j)$ denotes the Euclidean distance between two data points p_i and p_j.



(a) CNN search (b) CVNN search

Fig. 1 Example of CNN and CVNN queries

k nearest neighbor (VkNN) retrieval [4,5], which returns the $k(\geq 1)$ closest objects that are *visible* to a specified query point; (ii) *visible reverse k-nearest neighbor* search [6, 7], which retrieves the points in a data set P that have a given query point as one of their k visible nearest neighbors (VNNs), considering the blocks of obstacles in an obstacle set O; (iii) *obstructed nearest neighbor* (ONN) query [8,9], which finds the k points in a dataset that have the smallest *obstructed distances*[2] to a predefined query point; (iv) *continuous obstructed nearest neighbor* retrieval [10], which retrieves the ONN for every point along a specified query line segment according to the obstructed distance; and (v) *spatial clustering in the presence of obstacles* [11–16], which divides a set of two-dimensional data points into homogeneous groups (i.e., clusters) by taking the influence of obstacles into consideration. Nevertheless, most of the existing work only takes into account *fixed query points* instead of *moving query trajectories* that contain continuous query point locations. On the other hand, with the growing popularity of smart mobile devices and rapid advance of wireless technologies, more and more users issue queries even when they are moving. Consequently, the traditional *snapshot* query might not satisfy the real requirements from mobile users, and *continuous* query processing over a moving trajectory is required.

Based on these observations, in this paper, we investigate *continuous*[3] *visible nearest neighbor* (CVNN) search that finds the VNN of every point along a query line segment. To be more specific, given a data set P, an obstacle set O, and a query line segment q, a CVNN query retrieves the VNN for each point on q. It aims at finding a set of $\langle p, R\rangle$ tuples, where $p \in P$ is the VNN for any point in the interval $R \subseteq q$. It is important to note that p may be *empty*, meaning that all points in P are *invisible* to any point on R due to the obstruction of obstacles in O. Consider, for example, Figure 1(b), in which $P = \{a, b, c, d, f, g, h\}$, $O = \{o_1, o_2, o_3\}$ (denoted by shaded rectangles[4]), and $q = [s, e]$. The CVNN query returns $\{\langle a, [s, s_1]\rangle, \langle g, [s_1, s_2]\rangle, \langle c, [s_2, s_3]\rangle, \langle d, [s_3, e]\rangle\}$, which indicates that point a is the VNN for any point along interval $[s, s_1]$, point g is the VNN for any point along interval $[s_1, s_2]$, and so forth. Notice that point h is the NN for each point on interval $[s_2, s_3]$ in the conventional CNN retrieval as shown in Figure 1(a), whereas it is not the VNN for any point on $[s_2, s_3]$ in the CVNN search because of obstacle o_3.

In addition to the CVNN query introduced above, it has several interesting variations, including (i) *continuous visible k nearest neighbor* (CVkNN) search, which retrieves the k VNNs for every point on a given query line segment; (ii) *trajectory* VkNN (TVkNN) search, which returns the k VNNs of every point along an arbitrary trajectory consisting of multiple line segments; (iii) CVkNN query with *visible distance threshold* δ (δ-CVkNN) which, for each point p on a specified query line segment, finds the k nearest neighbors that are visible to p and meanwhile have their distances to p bounded by a given threshold δ; and (iv) *constrained* CVkNN (CCVkNN) search, which retrieves the k VNNs in the restricted area (defined by the spatial region constraints) for each point along a specified query line segment.

CVNN search and all these potential variants constitute a suite of interesting and practical problems from both the research point of view and application point of view. In this paper, we focus on CVNN retrieval because it not only introduces some new challenges but is also useful in many applications, such as decision support and location-based commerce. Two example applications are listed as follows.

Placement of traffic surveillance cameras. Suppose that Land Transport Authority (LTA) of Singapore wants to install traffic surveillance cameras to monitor accident-prone roads/streets[5]. Obviously, each location loc along the monitoring roads/streets should be *visible* to at least one camera c. In addition, the distance between location loc and its monitoring camera c is expected to be as small as possible in order to improve the video quality. By taking both visibility and distance into consideration, CVNN query can locate the best locations out of a given set of potential camera installation points for cameras to cover any point along the monitoring region[6].

Tourist recommendation. A CVkNN query can find out the k closest *visible* scenes (e.g., temple, stele, pagoda, etc.) for each (*sub*) route along a given tourist traveling route, defined by a starting point s and an ending point e. Different from conventional CNN retrieval, CVNN search considers all the physical obstacles such as buildings and mountains. Hence, the query result provides more accurate information

[2] The obstructed distance between any two data points in a data set is defined as the length of the shortest path that connects them without crossing any obstacle from a set of obstacles.

[3] Here, "continuous" denotes "continuously in spatial" instead of "continuously in time".

[4] Although an obstacle can be in any shape (e.g., triangle, pentagon, etc.), we assume it is a rectangle in this paper.

[5] We assume the monitoring roads/streets can be approximated by line segments.

[6] Note that although the placement of traffic surveillance cameras could be decided during offline planning process, efficient CVNN query processing algorithm is still preferred, given the fact that the number of monitoring regions considered and the number of traffic surveillance camera placement decision might be huge.

in terms of *visibility*. It is worth noting that, in this case, the purpose of CVNN query differs from that of *route query* which finds suitable routes that pass through part/all scenes included in a specified scene set, e.g., *optimal sequenced route* query [17, 18] and *trip planning* query [19].

Motivated by the significance of CVNN queries and the lack of efficient search algorithms, in this paper, we propose an efficient algorithm for processing CVNN retrieval and its variants. Our method (i) utilizes traditional data-partitioning indices (e.g., R-trees [20, 21]) on both the data set and the obstacle set, (ii) tackles *exact* CVNN search by performing a *single* index traversal, and (iii) enables a suite of effective pruning heuristics to only access the data points and obstacles *relevant to the final query result*. Moreover, the proposed CVNN search algorithm is general and can be easily extended to support different variations of CVNN queries, including CVkNN search, TVkNN search, δ-CVkNN search, and CCVkNN search. In summary, this paper has made five-fold contributions which are listed as follows:

- We formalize CVNN retrieval, a novel addition to the family of spatial queries, and reveal its unique characteristics. To the best of our knowledge, this paper is the first attempt on this problem.
- We propose a series of pruning heuristics on the data set and the obstacle set respectively to effectively prune those objects that do not contribute to the final query result and improve the search performance accordingly.
- We develop an efficient CVNN search algorithm, analyze its cost, and prove its correctness.
- We introduce several interesting variants of CVNN queries, and extend our techniques to handle them efficiently.
- We conduct extensive experiments using both real and synthetic datasets to demonstrate the effectiveness of our proposed pruning heuristics, and the performance of our proposed algorithms.

A preliminary report of this study appeared in [22]. We extend that work in this paper by (i) studying two new CVNN query variants, i.e., TVkNN search and CCVkNN search; (ii) evaluating the effectiveness of different pruning heuristics; and (iii) conducting a more comprehensive performance evaluation. Furthermore, we significantly improve the review of related work to make this paper self-contained.

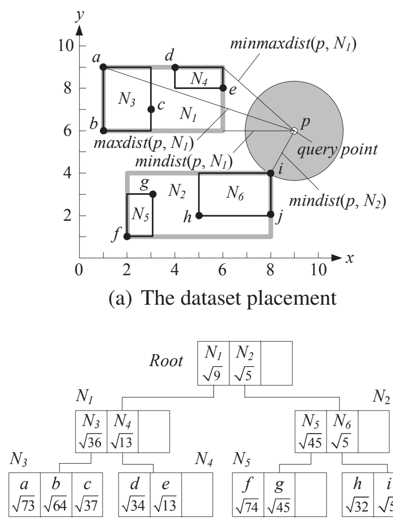
The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 formulates the problem and reveals its characteristics. Section 4 discusses the pruning heuristics on the data set P and the obstacle set O, respectively. Section 5 proposes efficient CVNN query processing algorithms, assuming that P and O are indexed by two *separate* R-trees and a *unified* R-tree, respectively. Section 6 extends our solution to deal with various variants of CVNN queries. Section 7 presents the performance study and reports our findings. Finally, Section 8 concludes the paper with some directions for future work.

2 Related work

In this section, we review the existing work related to CVNN queries, namely, NN search using R-trees, CNN retrieval, and visibility queries.

2.1 Algorithms for NN search on R-trees

R-tree [21] and its variants (e.g., R*-tree [20], etc.) are the most well-received spatial indexes due to their simplicity and efficiency. Figure 2 shows a data set $P = \{a, b, \cdots, j\}$ in a 2D space, and the corresponding R-tree assuming a capacity of *three* entries per node. Note that, in Figure 2(b), the number in each entry refers to the mindist between the query point p and the corresponding *Minimum Bounding Rectangle* (MBR) of the entry. As a leaf entry refers to a point p' in P, its mindist to p is the actual distance from p' to p. These numbers are not stored in R-tree previously, but computed *on-the-fly* during query processing.



(a) The dataset placement

(b) The R-tree

Fig. 2 Example of an R-tree and a NN query

An NN query finds the object in a dataset P that is the closest to a given query point p. Existing NN search algorithms traverse the R-tree on P in a *branch-and-bound* manner, and use some distance metrics, including mindist(p, N), maxdist(p, N), and minmaxdist(p, N), to prune the search space. Here, p is a query point and N is an R-tree node which corresponds to an MBR together with all the points covered. The mindist(p, N) and maxdist(p, N) provide the lower and upper bounds of the distances from p to any point in the subtree of N. The minmaxdist(p, N) defines an upper bound of the distance between p and its NN in N, that is, there is at least one point located inside N whose distance to p does not exceed minmaxdist(p, N). Figure 2(a) illustrates these pruning metrics between p and nodes N_1, N_2.

Existing algorithms for NN retrieval follow either *depth-first* (DF) or *best-first* (BF) traversal paradigm. DF algorithms [23,24] start from the root, and visit recursively the node with the smallest mindist to a given query point until the leaf level where a potential NN is found. Take an NN query issued at the point p shown in Figure 2(a) as an example, DF accesses *Root* first, followed by N_2, and then N_6, where the first NN candidate, i.e., point i, is discovered. Subsequently, the algorithm conducts backtracking operations. In particular, during backtracking to the upper levels, DF only visits those entries with minimum distances to p smaller than the distance between p and the NN candidate already retrieved. Continuing the above example, after finding i, DF backtracks to the root level (without visiting N_5 as $\mathsf{mindist}(p, N_5) > \mathsf{mindist}(p, i)$), where the NN candidate (i.e., i) is confirmed to be the actual NN of p. As demonstrated in [25], the DF algorithm is suboptimal, i.e., it accesses more nodes than necessary.

BF algorithms [26,27] achieve the optimal I/O performance by visiting only the nodes necessary for obtaining the NN. Towards this, BF maintains a priority queue (in this paper we use a heap H) with the entries visited so far, sorted in ascending order of their mindist to a specified query point p. Initially, BF inserts all the entries of the root into H (together with their mindist), e.g., in Figure 2, $H = \{(N_2, \sqrt{5})$, $(N_1, \sqrt{9})\}$. Then, at each step, BF visits the node in H with the minimal mindist. Continuing the running example, BF de-heaps the top N_2 of H, retrieves its content, and en-heaps all the entries, after which $H = \{(N_6, \sqrt{5}), (N_1, \sqrt{9}), (N_5, \sqrt{45})\}$. Similarly, the next node accessed is a leaf entry N_6, in which the data points are inserted into H $(= \{(i, \sqrt{5})$, $(N_1, \sqrt{9}), (j, \sqrt{17}), (h, \sqrt{32}), (N_5, \sqrt{45})\})$. Point i, the top of H, is taken as the current NN. At this time, the algorithm terminates with i as the final query result, because the next entry in H (i.e., N_1) is farther from p than i. Both DF and BF can be easily extended to retrieve k (> 1) nearest neighbors. Furthermore, BF is *incremental*, i.e., it returns the NNs in ascending order of their distances to the query point; and thus, k does not have to be known in advance.

In addition, different variants of NN queries have been investigated as well. Ferhatosmanoglu *et al.* [28] discuss *constrained* NN search that discovers the NN(s) in a restricted area of the data space. Papadias *et al.* [29,30] explore *aggregate* NN (and *group* NN) queries where, given a data set P and a query set Q, the goal is to retrieve the point(s) in P with the *smallest* aggregate (e.g., *sum*, *max*, *min*, etc.) distance(s) to all the points in Q. Zhang *et al.* [31] introduce *all* NN retrieval, which finds for each point $p_1 \in D_1$ its NN $p_2 \in D_2$, with D_1 and D_2 representing two specified datasets. Deng *et al.* [32] consider *surface* k-NN search, in which the distance is calculated from the *shortest path* along a terrain surface. Hu *et al.* [33] study the *range* NN query that returns the NN(s) for every point in a range.

2.2 CNN queries

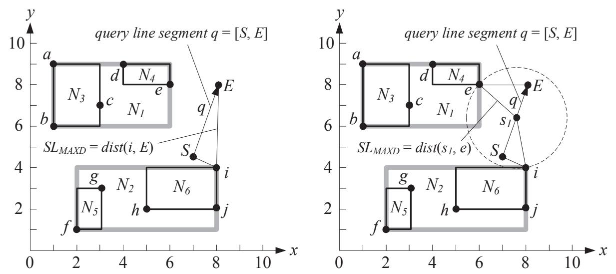
The CNN search has received considerable attention since it was first introduced by Sistla *et al.* [34] in the context of spatial-temporal databases. In that pioneering work, modeling methods and query languages for the expression of CNN queries are presented, but not the processing algorithms. The first algorithm for CNN query processing, based on *periodical sampling technique*, is proposed in [1]. Due to the natural disadvantage of sampling, its performance highly depends on the number and positions of sampling points, and the accuracy cannot be guaranteed. Therefore, the sampling based approach is not considered in this paper as it cannot tackle *exact* CVNN retrieval, the focus of this paper.

In order to conduct exact CNN search, two algorithms using R-trees are proposed in [2,3]. The first algorithm is based on the concept of *time-parameterized* (TP) queries, which treats a query line segment as the moving trajectory of a query point [2]. Hence, the nearest object to the moving query point is valid only for a limited duration, and a new TP query is issued to retrieve the next nearest object once the valid time of the current nearest object expires, i.e., when a split point is reached. Although the TP approach avoids the drawbacks of sampling, it needs to issue m TP queries with m being the number of answer objects[7]. In order to improve the performance, the second algorithm [3] finds all answer objects for the *whole* query line segment in a *single* round.

Since the algorithm proposed in this paper shares the same principle as CNN search proposed in [3], we illustrate the basic idea of CNN search using a running example. As shown in Figure 3, a CNN query is issued at line segment $q = [S, E]$, the straight line connecting S and E, with the data points depicted in Figure 2 forming a sample dataset P. The basic idea is to evaluate the data points in P according to the best-first order, i.e., those closer to q are evaluated earlier. For each evaluated point $p \in P$, it finds out the set of points along q that are *covered* by p, i.e., being closest to p, prunes away the points that will not cover any point along q, and fines tune the covering relationship during the traversal.

Initially, the result list is set to $\{\langle \emptyset, [S, E]\rangle\}$ which indicates that the whole query line segment is not covered by any point, and the pruning metric SL_{MAXD} that maintains the maximal distance between any point along q and its current NN object is set to ∞. Thereafter, the traversal of P starts. When point i, the first point accessed, is evaluated, it covers the whole query line segment. Consequently, the result list is updated to $\{\langle i, [S, E]\rangle\}$, and SL_{MAXD} is changed to $dist(i, E)$, as depicted in Figure 3(a). Next, e is evaluated. As it is closer to E than its current NN (i.e., i), the result list is updated to $\{\langle i, [S, s_1]\rangle, \langle e, [s_1, E]\rangle\}$ and SL_{MAXD} is decreased to the distance between s_1 and e, i.e., $dist(s_1, e)$

[7] For the rest of this paper, we refer to the data objects/points in the final query result as *answer objects/points*.



(a) After processing i (b) After processing e

Fig. 3 Example of CNN algorithm

as shown in Figure 3(b). Thereafter, j is evaluated. Since its minimal distance to q exceeds SL_{MAXD}, it will not invalidate the current covering relationship of any answer object and hence can be discarded safely. Here, the algorithm terminates because all the unexamined entries are guaranteed to have their minimal distances to q larger than SL_{MAXD}.

Based on the existing CNN search algorithms, a naive approach for answering CVNN query, namely *Baseline*, can be developed. The basic idea is to invoke CNN retrieval continuously to retrieve the NN objects, second NN objects, and so on until those visible to the specified query line segment q are found. Specifically, it first employs CNN search to locate the NN objects for q and then validates the visibility of answer objects. In case an answer object o is not visible (either completely or partially) to the line segment q' it covers, a new C2NN query has to be issued to find the next NN (i.e., 2nd NN) objects to q'. If the new object is visible to q', the search is completed. Otherwise, a new C3NN query has to be issued to retrieve the third NN object to q'. The routine proceeds until all the VNNs to q are identified. Given the fact that existing CkNN search returns kNN objects in a whole but not the k-th NN object, a C(k+1)NN query actually repeats all the efforts spent on a CkNN query. In order to support *incremental* CNN retrieval, Baseline preserves all the entries (data points and nodes) pruned away during CNN retrieval in an array ary to enable reuse and snapshots the min-heap hp when CNN search is completed. It inserts back the entries in ary into hp as an initial min-heap for the new CNN search. In other words, the Baseline guarantees a CVNN query can be answered via *multiple* CNN queries with *one* dataset traversal. However, it is still not efficient. First, it does not utilize visibility based pruning heuristics to discard those unqualified entries during the search. Second, it needs to conduct CNN search multiple times, resulting in high CPU overhead. Given the result list $\cup_i \langle o_i, q_i \rangle$ to a CVNN query, we assume the answer object o_i is actually the n_i-th NN object to any point along q_i. Baseline in total has to invoke $MAX_i(n_i)$ CNN queries. If we further improve the performance by starting the Ck_1NN query with $k_1 > 1$ and then increasing the value of k_1 by r instead of 1 thereafter, the number of CNN queries performed could be reduced to $(MAX_i(n_i) - k_1)/r + 1$. Nevertheless, how to select the values of k_1 and r is challenging.

In addition, some variations of CNN search have been proposed in the literature. Iwerks *et al.* [35] study *continuous windowing algorithm* to answer CkNN retrieval via less expensive range queries. However, the algorithm is only sub-optimal when the location updates of moving objects are frequent or the k value is large. In view of this, Li *et al.* [36] develop a *beach-line algorithm*, which monitors only the k-th NN to maintain the CkNN query result, instead of monitoring all k NNs.

Recently, the CNN *monitoring* problem that monitors the answer objects to a CNN query for a given duration, has been studied. Different monitoring algorithms (e.g., CPM [37], SEA-CNN [38], and YPK-CNN [39]) have been proposed, based on the concept of *monitoring region*. Here, the monitoring region corresponding to a query q refers to an area inside which the movement of objects might affect the query result, and hence those objects that are always outside the region could be safely discarded. Other versions of CNN monitoring include (i) CNN monitoring in the road network [40,41], where the distance between any two objects is defined as the length of their shortest path; and (ii) CNN monitoring in the distributed environment [42, 43], where the optimization target is to reduce the communication cost between the central query processor and the data objects. More recently, Zheng *et al.* [44] investigate CNN retrieval in wireless data broadcast systems, where mobile clients answer their own CNN queries by listening to the wireless broadcast channel. In addition, CNN retrieval in spatial network databases has been studied in [45–47].

All the aforementioned work on CNN search and its variants do not consider obstacles that exist in many real-life scenarios. Consequently, existing algorithms for them cannot be applied to handle CVNN retrieval efficiently.

2.3 Visibility queries

Although visibility computation algorithms have been well-studied in the area of computer graphics and computational geometry [48], there are only a few works on *visibility queries* in the database community [49–51]. The existing methods utilize various indexing structures (e.g., LoD-R-tree [49], HDoV-tree [51], etc.) to deal with visibility queries in visualization systems. Since these specialized access methods are designed only for the purpose of visualization without maintaining any distance information, they are not capable of supporting efficient CVNN query processing.

Recently, Nutanong *et al.* [4,5] introduce *visible nearest neighbor* (VNN) search to find the NN that is *visible* to a specified query point. An example of VNN query issued at s_4 is depicted in Figure 1(b). The answer point is d. Although point h is closer to s_4 than d, it is blocked by obstacle

o_3 and hence is excluded from the final query result. A VNN query algorithm, based on the fact that a farther object cannot affect the visibility of a nearer object, is proposed in [4, 5]. The basic idea is to perform NN search and check its visibility condition in an incremental manner. Nevertheless, the algorithm is only for a fixed query point, but not a line segment which contains multiple query points.

In our earlier work [6, 7], we have investigated *visible reverse nearest neighbor* (VRNN) search where, given a data set P, an obstacle set O, and a query point q, the goal is to retrieve the points in P that have q as their VNN. We propose an efficient algorithm for VRNN query processing, assuming that both P and O are indexed by R-trees. Our solution follows a *filter-refinement* framework, and requires *no* pre-processing. Specifically, a set of candidate objects (i.e., a superset of the final query result) is found in the filter step, and gets refined in the subsequent refinement step, with these two steps integrated into a *single* R-tree traversal. As the size of the candidate objects has a direct impact on the search efficiency, we employ *half-plane properties* (as [52]) and *visibility check* to prune the search space.

Based on the visibility query, we can employ a *brute force based algorithm* (BFA) to answer CVNN search. It first invokes visibility test to evaluate each and every data point p in a given data set P, and then examines whether p is closer to any point along the query line segment q than its current NN object if p is visible either *partially* or *completely* to q. Obviously, BFA suffers from the blind and exhaustive scanning as it does not utilize any pruning technique and has to scan the entire dataset in sequence. The experimental results to be reported in Section 7 will further demonstrate its poor performance. Note that, although BFA could be improved via pre-computing object visibility, we leave the investigation of the improved BFA to our future work due to the limitation of space.

3 Preliminaries

In this section, we first present problem definitions for CVNN search, and then reveal some unique characteristics that can facilitate the development of efficient CVNN query processing algorithms. Table 1 summarizes the symbols used in the rest of this paper.

3.1 Problem definitions

Given a set of data points $P = \{p_1, p_2, \cdots, p_n\}$, a set of obstacles $O = \{o_1, o_2, \cdots, o_m\}$, and a query line segment $q = [s, e]$ in a two-dimensional (2D) space, visibility between two points p, p' is defined in Definition 1, based on which we formulate VNN and CVNN queries in Definition 3 and Definition 4, respectively.

Table 1 Symbols and descriptions

Notation	Description
P	A set of data points p in a two-dimensional space
O	A set of obstacles o in a two-dimensional space
T_p	The R-tree on P
T_o	The R-tree on O
q	A query line segment with $q = [s, e]$
R	An interval of q with $R = [R.l, R.r]$ $(\subseteq q)$
RL	The result list of a CVNN query
L_o	The linked list storing obstacles
$\perp (p, p')$	The perpendicular bisector of the line segment $[p, p']$
R_c	Constrained region

Definition 1 Visibility. Given $p, p' \in P$, p and p' are *visible* to each other iff there is *no* any obstacle o in O such that the straight line connecting p and p', denoted by $[p, p']$, crosses o, i.e., $\forall o \in O$, $o \cap [p, p'] = \emptyset$.

Definition 2 Visible region. Given $p \in P$ and q, the *visible region* of p over q, denoted by VR_p, is defined as the set of intervals $R \subseteq q$ such that p is visible to all points in R.

Definition 3 Visible nearest neighbor [4]. Given $p' \in P$ and $p \notin P$, p' is the *visible nearest neighbor* (VNN) of p iff: (i) p' is visible to p; and (ii) $\forall p'' \in P - \{p'\}$, if p'' is visible to p, $dist(p'', p) \geq dist(p', p)$.

Definition 4 Continuous visible nearest neighbor query. Given P, O, and q, a *continuous visible nearest neighbor* (CVNN) *query* returns a result list RL that contains a set of $\langle p_i, R_i \rangle$ $(i \in [1, t])$ tuples such that (i) $\forall i, j \in [1, t] (i \neq j)$, $R_i \cap R_j = \emptyset$[8]; (ii) $\cup_{i=1}^{t} R_i = q$; and (iii) $\forall \langle p_i, R_i \rangle \in RL$, if $p_i \neq \emptyset$, p_i is the VNN of any point along R_i.

Definition 5 Dominance. Given $p \in P$ and R, p *dominates* R iff $\forall p' \in P - \{p\}$ and any point r along R (i.e., $\forall r \in R$), $dist(p, r) \leq dist(p', r)$.

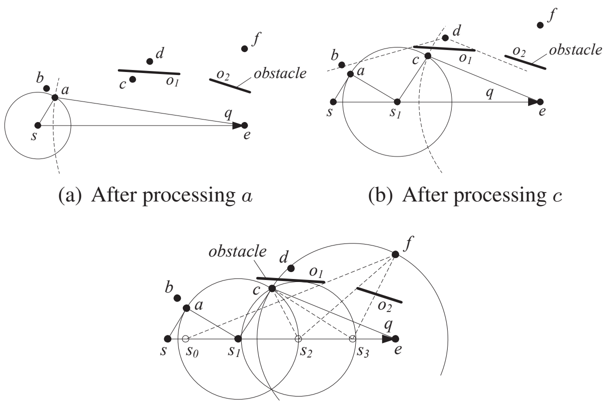
Definition 6 Dominated region. Given $p \in P$ and q, the *dominated region* of p over q, denoted by DR_p, is defined as the set of intervals $R \subseteq q$ that are dominated by p.

To illustrate the concept of dominance, Figure 4(a) depicts an example, in which $P = \{a, b\}$ and $R = [s, e]$ (i.e., q). As $dist(b, s) > dist(a, s)$ and $dist(b, e) > dist(a, e)$, it is certain that a is closer to any point along q, compared with b. Hence, point a dominates q.

Suppose an interval $R = [R.l, R.r]$ is dominated by a point p, we define the circle $cir(R.l, p)$ ($cir(R.r, p)$) that centers at $R.l$ ($R.r$) and has $dist(p, R.l)$ ($dist(p, R.r)$) as the radius as the *vicinity circle* of $R.l$ ($R.r$), denoted by $VC(R.l)$ ($VC(R.r)$). Any other point p' that can *violate* p's dominance over R must be within either $VC(R.l)$ or $VC(R.r)$, as to be demonstrated in Lemma 1. Back to the

[8] If R_i and R_j are *adjacent*, i.e., $|i - j| = 1$, $R_i \cap R_j \neq \emptyset$.

above example. Assume a new point c is added into P, and it violates a's dominance on q (= $[s, e]$) since c is inside e's vicinity circle, i.e., $VC(e)$ centered at e with $dist(a, e)$ as radius. The appearance of point c actually partitions the interval q into two sub-intervals R_1 (= $[s, s_1]$) and R_2 (= $[s_1, e]$), with a dominating R_1 and c dominating R_2 respectively, as shown in Figure 4(b). Point s_1 is defined as the *split point*, i.e., the point on the interval where the VNN changes.



(a) After processing a (b) After processing c

(c) After processing f

Fig. 4 Updating result list

3.2 Problem characteristics

According to Definition 4, we understand that CVNN search takes into account both the *proximity* and *visibility* between the data points and the query line segment. Thus, we develop Lemma 1 and Lemma 2 to facilitate the proximity checking and visibility checking, respectively. Then, Lemma 3 summarizes the condition that a VNN object must satisfy.

Lemma 1 *Assume point p dominates an interval $R = [R.l, R.r]$. A new point p'* violates *p's dominance over R iff p' is within $VC(R.l)$ or $VC(R.r)$, i.e., $p' \in VC(R.l) \cup VC(R.r)$.*

Proof We first proof sufficiency. If p' is within $VC(R.l)$, $dist(p', R.l) < dist(p, R.l)$ and hence p' violates the dominance of p over R. Similarly, if p' is inside $VC(R.r)$, $dist(p', R.r) < dist(p, R.r)$ and thus p' violates p's dominance on R. We now prove necessity. If p' violates the dominance of p over R, it means there is at least one point p'' along R such that $dist(p'', p) > dist(p'', p')$. In other words, $p' \in cir(p'', p)$, i.e., point p' must be within the vicinity circle that centers at p'' and has $dist(p'', p)$ as the radius. However, according to the geometric knowledge, $cir(p'', p) \subseteq VC(R.l) \cup VC(R.r)$. Therefore, $p' \in cir(p'', p)$ indicates $p' \in VC(R.l) \cup VC(R.r)$. The proof completes. □

Lemma 2 *Given an interval $R = [R.l, R.r]$ and a new data point p, p will not be the VNN of any point along R if p is* invisible *to every point in R.*

Proof The proof is obvious because the data point p that is the VNN of R (i.e., p is the VNN of every point along R) must be *visible* to each point in R. □

Lemma 3 *Point p must be the VNN of any point along interval $R = VR_p \cap DR_p$.*

Proof According to Definition 2, VR_p is the visible region of p, meaning that p is visible to any point in VR_p. According to Definition 6, DR_p is the dominated region of p, indicating that p dominates DR_p, that is, p is the NN to every point in DR_p. Consequently, p must be the VNN of any point along interval R ($= VR_p \cap DR_p$) by Definition 3. □

Lemma 1 suggests an incremental query processing approach, which aims at reporting the result of CVNN retrieval issued at a given query line segment $q = [s, e]$ with a *single* dataset traversal. Initially, result list RL is set to $\{\langle \emptyset, [s, e] \rangle\}$, meaning that currently the VNNs of all the points in $[s, e]$ are unknown. Thereafter, we evaluate the impact of a new point p on RL by checking whether p is located inside the vicinity circle of $R_i.l$ or $R_i.r$ with respect to a tuple $\langle p_i, R_i \rangle \in RL$. If p violates the dominance of an answer object p_i on the interval R_i, the RL is updated. The evaluation continues until all the points in the dataset P are examined.

Figure 4 depicts a running example with dataset $P = \{a, b, c, d, f\}$, obstacle set $O = \{o_1, o_2\}$[9], and query line segment $q = [s, e]$. Here, points in P are processed in alphabetic order. At the beginning, RL is set to $\{\langle \emptyset, [s, e] \rangle\}$. As a is the first point encountered and its view is not blocked by any obstacle in O, it becomes the current VNN of each point in q, i.e., $RL = \{\langle a, [s, e] \rangle\}$. Second, point b is evaluated. We only need to check whether b falls into $VC(s)$ or $VC(e)$ (i.e., whether b is closer to s or e than its current VNN). The fact that b is outside both vicinity circles guarantees that b does not dominate any point along $[s, e]$ and thus b is discarded.

Next, point c is checked. Since c is inside $VC(e)$ and it is visible to every point in $[s, e]$, a split point s_1 is created. It is the intersection between the query line segment (i.e., $[s, e]$) and the perpendicular bisector of the line segment $[a, c]$ (i.e., $\perp (a, c)$), indicating that points to the left of s_1 are closer to a while points to the right of s_1 are closer to c. Consequently, RL is updated to $\{\langle a, [s, s_1] \rangle, \langle c, [s_1, e] \rangle\}$. Figure 4(b) depicts the case after the processing of point c. Then, point d is evaluated and gets pruned because it is not visible to any point along q, although d violates c's dominance on $[s_1, e]$ (see Figure4(b)). Finally, point f is examined. It does not contribute to the CVNN query result as its visible region VR_f (= $[s_0, s_2]$) and dominated region DR_f (= $[s_3, e]$) are disjoint. After the processing of f, as shown in Figure 4(c),

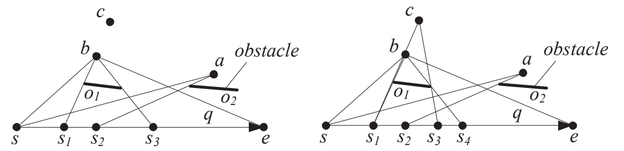
[9] To simplify the discussion, we use line segments, but not rectangles, to represent obstacles in the rest of this paper, while our methods can be used with rectangles that are sets of line segments.

the final query result $RL = \{\langle a, [s, s_1]\rangle, \langle c, [s_1, e]\rangle\}$ is retrieved and the CVNN search is terminated.

In addition, we observe two important properties, namely, *VNN discontinuity* and *invisible interval*, which are unique to the CVNN query.

Property 1 VNN discontinuity. *A data point p may be the VNN to multiple intervals that are not adjacent.*

For instance, Figure 5(a) depicts a situation where data points a, b have been processed, and the corresponding RL = $\{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle \emptyset, [s_2, s_3]\rangle, \langle b, [s_3, e]\rangle\}$. Point b is the VNN for all the points along intervals $[s, s_1]$ and $[s_3, e]$ that are not *adjacent*. This property implies that a binary search heuristic, which is used in conventional CNN search to retrieve the dominated region for a specified point, cannot be applied to CVNN retrieval.



(a) VNN discontinuity (b) Invisible interval

Fig. 5 Illustration of problem properties

Property 2 Invisible interval. *The result list RL of a CVNN query may have $k(\geq 1)$ invisible intervals $\langle \emptyset, R\rangle$, where no point in a given dataset is visible to any point in R.*

Continuing the running example, Figure 5(b) illustrates the situation after the processing of point c, in which RL = $\{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle \emptyset, [s_2, s_3]\rangle, \langle c, [s_3, s_4]\rangle, \langle b, [s_4, e]\rangle\}$. In this case, $[s_2, s_3]$ is an *invisible interval*.

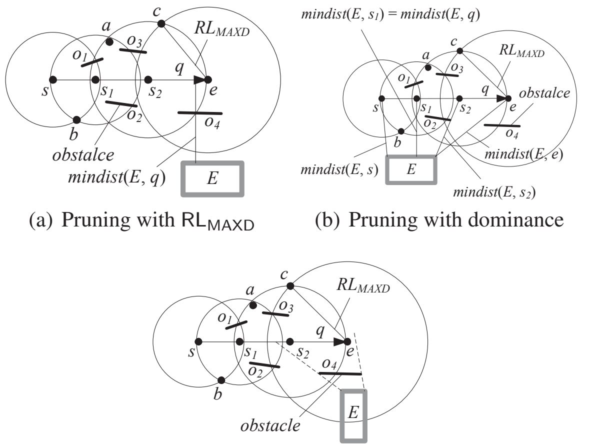
4 Pruning heuristics

We adopt *branch-and-bound* techniques to process CVNN queries. In order to prune the search space, a series of pruning heuristics are developed. In this section, we explain the detailed pruning heuristics for data set P and obstacle set O, respectively.

4.1 Pruning on data set

Heuristic 1 *Suppose the current result list $RL = \cup_{1\leq i\leq t}$ $\langle p_i, R_i\rangle$, with $R_i = [R_i.l, R_i.r]$. Given an intermediate node entry E and a query line segment q, the subtree of E may contain some answer points only if* $\mathsf{mindist}(E, q) < \mathsf{RL_{MAXD}}$, *where* $\mathsf{mindist}(E, q)$ *denotes the minimum distance from the MBR of E to q, and* $\mathsf{RL_{MAXD}} = MAX_{1\leq i\leq t}\ (dist(p_i, R_i.l), dist(p_i, R_i.r))$.

Figure 6(a) shows a data set $P = \{a, b, c\}$, an obstacle set $O = \{o_1, o_2, o_3, o_4\}$, a query line segment $q = [s, e]$, and current $RL = \{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle c, [s_2, e]\rangle\}$. Rectangle E represents the MBR of an intermediate (i.e., a non-leaf) node. As $\mathsf{mindist}(E, q) > \mathsf{RL_{MAXD}} = dist(c, e)$, E does not contain any point that dominates some interval of q, and hence the search space covered by E can be safely pruned. Note that the calculation of mindist between a rectangle (i.e., MBR) E and a line segment q, i.e., $\mathsf{mindist}(E, q)$, is presented in [3].



(a) Pruning with $\mathsf{RL_{MAXD}}$ (b) Pruning with dominance

(c) Pruning with visibility

Fig. 6 Pruning techniques

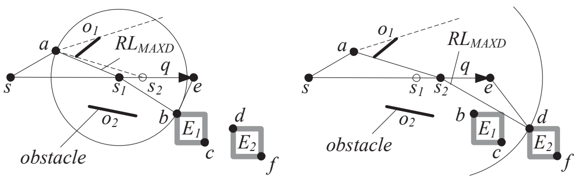
Heuristic 1 can serve as the initial pruning criteria since its computational overhead is very small. However, an entry E with $\mathsf{mindist}(E, q) < \mathsf{RL_{MAXD}}$ does not necessarily contain any answer object, which means that the pruning condition can be improved further. To verify this, consider Figure 6(b), which is similar to Figure 6(a) except that $\mathsf{RL_{MAXD}}$ is larger. Notice that although E cannot be pruned by Heuristic 1 as $\mathsf{mindist}(E, q)$ (= $\mathsf{mindist}(E, s_1)$) $< \mathsf{RL_{MAXD}}$, E does not contain any qualified data point that dominates a certain interval of q. Consequently, Heuristic 2 is devised to prune away such entries.

Heuristic 2 *Given an intermediate node entry E and a query line segment q, the subtree of E may contain answer points only if there is at least one interval R in RL such that* some *points on R are dominated by E.*

Heuristic 2 gives a stronger pruning criterion, but it incurs higher CPU cost compared with Heuristic 1, because it requires the calculation of the minimal distance from E to each interval included in the current RL. Therefore, it is applied only for the entries that cannot be pruned away by Heuristic 1. Nevertheless, the access to entries satisfying both Heuristic 1 and Heuristic 2 is not always necessary. Take Figure 6(c) as an example. E satisfies Heuristic 1 and Heuristic 2, but it can be pruned away because it is *invisible* to $[s_2, e]$ due to the obstruction of obstacle o_4. Heuristic 3 enables this pruning.

Heuristic 3 *Given an intermediate node entry E and a query line segment q, the subtree of E needs to be accessed if there is an interval R in RL such that (i) $\exists R' \subseteq R$, R' is completely dominated by E; and (ii) E is visible to any point along R'.*

By taking the visibility into consideration, Heuristic 3 further eliminates unqualified entries, whereas it also incurs higher CPU overhead. Thus, it is utilized only for the entries that cannot be pruned by both Heuristic 1 and Heuristic 2.



(a) Processing E_1 first (b) Processing E_2 first

Fig. 7 Sequence of entry accesses

In addition, the entry access order plays an important role as well. As an example, consider Figure 7, in which point a has been processed, but not entries E_1 and E_2. The current $RL = \{\langle a, [s, s_2]\rangle, \langle \emptyset, [s_2, e]\rangle\}$, and $\mathsf{RL_{MAXD}} = \infty$. Since both E_1 and E_2 cannot be pruned by Heuristic 1, Heuristic 2, and Heuristic 3, they are accessed. Suppose that E_1 is visited first, then data points b, c in its subtree are processed. RL is updated to $\{\langle a, [s, s_1]\rangle, \langle b, [s_1, e]\rangle\}$, as shown in Figure 7(a). Thereafter, E_2 can be pruned away from further exploration by Heuristic 1. On the other hand, if E_2 is accessed first, $RL = \{\langle a, [s, s_2]\rangle, \langle d, [s_2, e]\rangle\}$ and E_1 has to be visited (see Figure 7(b)). To minimize the number of node accesses, we propose the following visiting order heuristic, which is based on the intuition that entries closer to the query line segment are more likely to contain qualifying data points.

Heuristic 4 *Entries E are accessed in a* best-first *fashion according to the ascending order of their* mindist *to the query line segment q.*

4.2 Pruning on obstacle set

A line segment q in a 2D space can divide the data space into two half-planes, as defined in Definition 7.

Definition 7 Half-plane. Given a query line segment q in a two-dimensional space, the data space is partitioned by q into two *half-planes*: $HP_q^{\perp}$ that is above q, and $HP_q^{\top}$ that is below q.

Observe that if a data point p lies in plane $HP_q^{\top}$ ($HP_q^{\perp}$), i.e., $p \in HP_q^{\top}$ ($p \in HP_q^{\perp}$), only those obstacles that overlap the half-plane $HP_q^{\top}$ ($HP_q^{\perp}$) could affect p's visibility with respect to q. For instance, as shown in Figure 8, the obstacles affecting the visibility of point a include o_1 and o_3; and the obstacles affecting c's visibility contain o_2 and o_3. Based on this observation, we propose the obstacle distribution heuristic below.

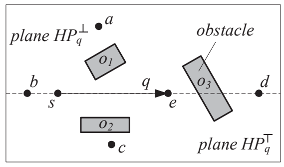
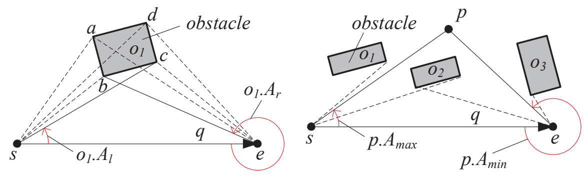


Fig. 8 Pruning with obstacle distribution

Heuristic 5 *Given a data point p and a query line segment q, an obstacle o that may affect the visibility of p with respect to q must overlap the half-plane partitioned by q that contains p, denoted as $HP_p(q)$.*

Heuristic 6 *Given a data point p and a query line segment $q = [s, e]$, the obstacles may affect the visibility of p with respect to q if they overlap the triangle formed by p and q, denoted as $\triangle pse$.*



(a) The $o_1.A_l$ and $o_1.A_r$ (b) The $p.A_l$ and $p.A_r$

Fig. 9 Pruning with angular domain

Given a data point p and a query line segment $q = [s, e]$, Heuristic 6 indicates that any obstacle o with $o \cap \triangle pse = \emptyset$ can be discarded because it has *zero* impact on p's visibility with respect to q. Therefore, we can reduce significantly the number of obstacles that need evaluations by applying Heuristic 6.

Next, we explain how to determine whether an obstacle shares some common area with $\triangle pse$. Our method is as follows. For a new obstacle o, we compute in *counter-clockwise* direction its minimum (maximum) angle, denoted by $o.A_s$ ($o.A_e$), between a specified query line segment q and the line segments connecting the starting (ending) point s (e) of q and the vertexes of o. For instance, an example is depicted in Figure 9(a), where $o_1.A_s = \angle cse$ and $o_1.A_e = \angle seb$. When processing a candidate data point p, we first calculate in *counter-clockwise* direction its minimum (maximum) angle, denoted by $p.A_s$ ($p.A_e$), formed by the query line segment q and the line segment connecting p and the starting (ending) point s (e) of q. Thereafter, any obstacle o that satisfies $o.A_s > p.A_s$ or $o.A_e < p.A_e$ does not need to be processed since it cannot intersect or locate inside $\triangle pse$. Consider, for example, Figure 9(b), in which $p.A_s = \angle pse$ and $p.A_e = \angle sep$; and hence, obstacle o_2, but not obstacles o_1 and o_3, affects p's visibility with respect to q.

Heuristic 7 *Given a data point p and a query line segment q, any obstacle o may affect the visibility of p with respect to q only if* $\mathsf{mindist}(o, q) < \mathsf{mindist}(p, q)$.

Clearly, Heuristic 7 is correct. According to Heuristic 6, all the obstacles that can affect the visibility of a data point p with respect to a given line segment $q = [s, e]$ must overlap the triangle formed by p and $[s, e]$, i.e., $\triangle pse$. On the other hand, the minimal distance between any point p' located inside $\triangle pse$ and q (i.e., $\mathsf{mindist}(p', q)$) is smaller than or equal to $\mathsf{mindist}(p, q)$. Consequently, if the obstacle o satisfies $\mathsf{mindist}(o, q) > \mathsf{mindist}(p, q)$, it must be located outside $\triangle pse$ and thus it does not affect p's visibility.

It is worth noting that our proposed pruning heuristics only target at a two-dimensional space, and they do not necessarily hold in three or higher dimensional spaces. Although it is challenging and interesting to develop effective pruning heuristics for CVNN search in a high-dimensional space, we leave it to our future work due to the focus of this work and the space limitation.

5 CVNN query processing

In this section, we present an efficient algorithm for CVNN search, assuming that the data set P and the obstacle set O are indexed by two *separate* R-trees. The basic idea is to traverse points in P according to ascending order of their mindist to a given query line segment $q = [s, e]$ (as implied Heuristic 4). For each data point $p \in P$ visited, we need to check whether p will update the current result list RL that is initialized to $\{\langle \emptyset, [s, e] \rangle\}$. To be more specific, we need to evaluate whether p violates the dominance of any existing answer object p_i on an interval R_i (either partially or completely), with $\langle p_i, R_i \rangle \in RL$.

In the following, we first present three sub-tasks involved in this evaluation: (i) how to find out all the obstacles that may affect the visibility of p, (ii) how to identify the visible region of p (i.e., VR_p) on q in the presence of obstacles, and (iii) how to evaluate p's impact on the current RL and how to do the update, in Sections 5.1, 5.2, and 5.3, respectively. Then, we propose the complete CVNN search algorithm in Section 5.4, together with the analysis of its time complexity and the proof of its correctness. Finally, we discuss how to adjust the search algorithm to tackle the CVNN query when dataset P and obstacle set O are indexed by one *unified* R-tree in Section 5.5.

5.1 Obstacle retrieval

In order to derive the visible region of point $p \in P$, we have to get all the obstacles in O that may affect p's visibility on q. The solution, namely *Get Obstacle Algorithm* (GetObs), is presented in Algorithm 1. The main idea is to scan the obstacle set O based on ascending order of the distances between the obstacles and the query segment q. According to Heuristic 7, only obstacles $o \in O$ with $\mathsf{mindist}(o, q) \leq \mathsf{mindist}(p, q)$ need to be evaluated, and thus the traversal on O can be safely terminated once the accessed obstacle has its distance to q larger than a specified search distance r, which is set to $\mathsf{mindist}(p, q)$ with p is the data point currently under evaluation. The result obstacles are stored in a linked list L_o.

Algorithm 1 Get Obstacle (GetObs)

Input: an obstacle R-tree T_o; a min-heap H_o; a search distance r; a query line segment q; a linked list L_o storing obstacles

while H_o is not empty **do**
  de-heap the top entry e of H_o
  if $\mathsf{mindist}(e, q) > r$ **then** // use Heuristic 7
    return L_o
  else if e is an obstacle **then**
    add e to L_o
  else // e is an intermediate node
    for each child entry $e_i \in e$ from T_o **do**
      insert e_i into H_o

In addition, we would like to highlight that since points in P are examined in ascending order of their distances to q, GetObs, for a point $p \in P$, does not need to *start from scratch*. Suppose $p_2 \in P$ is examined right after $p_1 \in P$. As $\mathsf{mindist}\ (p_1, q) \leq \mathsf{mindist}(p_2, q)$, all the obstacles that might affect p_1's visibility, denoted as GetObs(p_1), also have the possibility to affect p_2's visibility. Assume the obstacle list L_o returned by GetObs(p_1) is locally available, GetObs corresponding to p_2 only needs to retrieve those obstacles having distances to q between $\mathsf{mindist}(p_1, q)$ and $\mathsf{mindist}(p_2, q)$. In general, GetObs corresponding to a data point $p_{i+1} \in P$ that is examined right after data point $p_i \in P$ only needs to find out all the obstacles having their distances to q falling inside the range $[\mathsf{mindist}(p_i, q),\ \mathsf{mindist}\ (p_{i+1}, q)]$. Hence, GetObs is an *incremental* process, and it can find obstacles, for all the data points in P, via *one* traversal of O.

As an example, Figure 10 illustrates the incremental process of GetObs algorithm. In particular, GetObs is first invoked to obtain the obstacle o_1 that may influence the visibility of point a, maintained in L_o (i.e., $L_o = \{o_1\}$). Then, GetObs is called again for data point b. Since all the obstacles in current L_o might affect b's visibility on q, GetObs only needs to find out all the obstacles other than those in L_o (i.e., o_2 and o_3), after which L_o is updated to $\{o_1, o_2, o_3\}$. If there is a new data point (e.g., c) visited after b, all the obstacles in L_o can be reused, and the search on O can be continued to get the rest of the obstacles (e.g., o_4) that may affect its visibility.

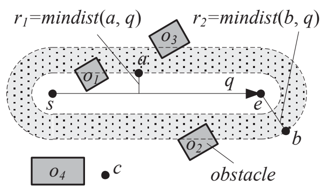


Fig. 10 Incremental access of obstacles

5.2 Visible region computation

Once all the obstacles that might affect the visibility of point $p \in P$ are retrieved via GetObs and maintained by L_o, we can identify the *invisible region* of p over q that is blocked by obstacle $o \in L_o$, denoted as $IR_{p,o}$. Then, p's *visible region* VR_p on q can be easily derived based on $VR_p = q - \cup_{o \in L_o} IR_{p,o}$.

Algorithm 2 Visible Region Computation (VRC)

Input: a data point p; a query line segment $q = [s, e]$; a linked list L_o that maintains obstacles
Output: p's visible region VR_p over q

1: $VR_p = q$
2: **for** each obstacle $o \in L_o$ **do**
3: **if** $o \cap HP_p(q) \neq \emptyset$ and $o \cap \triangle pse \neq \emptyset$ **then** // Heuristics 5, 6
4: $IR_{p,o} =$ IRC(q, p, o)
5: **for** each region $[l, r] \in IR_{p,o}$ **do**
6: $VR_p = VR_p - [l, r]$
7: return VR_p

Based on this basic idea, Algorithm 2 depicts the pseudo-code of the *Visible Region Computation Algorithm* (VRC). It takes as input a data point p, a query line segment $q = [s, e]$, and a linked list L_o that maintains all the obstacles affecting the visibility of p on q, and outputs p's visible region VR_p over q. VRC utilizes Heuristic 5 and Heuristic 6, and only evaluates those obstacles $o \in L_o$ which share some common area with the half-plane $HP_p(q)$ and meanwhile overlap with the triangle $\triangle pse$, to form the visible region for a given data point p. The function IRC(q, p, o) invoked in line 4 of Algorithm 2 is to return the regions inside q that are *invisible* to p due to the obstruction of obstacle o.

We illustrate Algorithm 2 using the example shown in Figure 11, where the obstacles affecting the visibility of p are maintained in $L_o = \{o_1, o_2, o_3\}$. VRC initializes VR_p to $q = [s, e]$ and recursively evaluates each obstacle in L_o. Specifically, it first examines obstacle $o_1 \in L_o$ and gets p's invisible region on q blocked by o_1, i.e., $IR_{p,o_1} = [s_1, s_3]$. Consequently, VR_p is updated to $q - IR_{p,o_1} = \{[s, s_1], [s_3, e]\}$. Next, VRC checks obstacle $o_2 \in L_o$ and obtains p's invisible region over q obstructed by o_2, i.e., $IR_{p,o_2} = [s_2, s_4]$, after which VR_p is updated to $\{[s, s_1], [s_4, e]\}$. Finally, obstacle o_3 is evaluated. Since p's invisible region along q blocked by o_3, i.e., IR_{p,o_3}, is $[s_5, s_6]$, VR_p is updated to $\{[s, s_1], [s_4, s_5], [s_6, e]\}$. VRC outputs $\{[s, s_1], [s_4, s_5], [s_6, e]\}$ as the final p's visible region on q to terminate the visible region computation for point p.

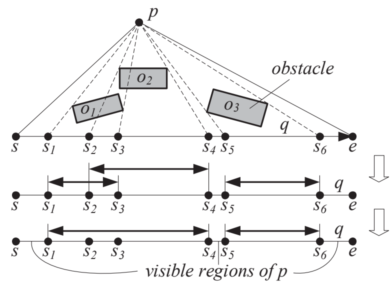


Fig. 11 Example of VRC algorithm

5.3 Result list update

For a data point $p \in P$ that is currently under evaluation, once its visible region on q (i.e., VR_p) is formed via VRC, we need to evaluate the impact of p on the current result list RL. Towards this, a *Result List Update Algorithm* (RLU) is developed to incrementally update RL for a CVNN query upon the evaluation of p. It takes the current result list $RL = \cup_{i=1}^{t} \langle p_i, R_i \rangle$, point p, and p's visible region VR_p as input, and outputs the updated result list.

As depicted in Algorithm 3, it performs the update via scanning every tuple $\langle p_i, R_i \rangle$ in RL. If p is *visible* to R_i either partially or completely (i.e., $R_i \cap VR_p \neq \emptyset$), RLU first derives the intersection R_{int} (= $R_i \cap VR_p$) and difference R_{dif} (= $R_i - R_{int}$) between R_i and VR_p. Thereafter, if the VNN of R_i is empty (i.e., $p_i = \emptyset$), $\langle p, R_{int} \rangle$ and $\langle \emptyset, R_{dif} \rangle$ (if $R_{dif} \neq \emptyset$) are inserted into a temporary result list TRL. Otherwise (i.e., $p_i \neq \emptyset$), RLU inserts $\langle p_i, R_{dif} \rangle$ into TRL if $R_{dif} \neq \emptyset$; and then, the algorithm invokes RS-CVNN algorithm to determine whether p_i can be partially/completely replaced by p over region R_{int}. On the other hand, p may be *invisible* to R_i (i.e., $R_i \cap VR_p = \emptyset$) and hence p has a *zero* impact on region R_i. RLU inserts $\langle p_i, R_i \rangle$ into TRL. After all the tuples in RL are evaluated, it outputs TRL as the updated result list. It is important to note that whenever a new tuple $\langle p', R' \rangle$ is inserted into TRL, it might be merged with an existing region R'' in TRL if R' and R'' are continuous and they share the same VNN, with the merge operation represented by Merge().

The RS-CVNN algorithm is used to check whether R_{int}'s current VNN p_i is still valid upon the existence of p, and replace p_i with p either partially or fully if necessary. The pseudo-code is described in Algorithm 4. Note that the region R_{int} = $[l, r]$ ($\subseteq VR_p$) is certainly visible to p, and

Algorithm 3 Result List Update (RLU)

Input: a result list RL; a data point p; p's visible region VR_p
Output: the updated result list

$TRL = \{\langle\emptyset, [s, e]\rangle\}$
for each tuple $\langle p_i, R_i\rangle \in RL$ **do**
  if $R_i \cap VR_p \neq \emptyset$ **then**
    $R_{int} = R_i \cap VR_p$ and $R_{dif} = R_i - R_{int}$
    if $p_i = \emptyset$ **then**
      insert $\langle p, R_{int}\rangle$ into TRL, insert $\langle\emptyset, R_{dif}\rangle$ into TRL if $R_{dif} \neq \emptyset$, and Merge() if necessary
    else
      add $\langle p_i, R_{dif}\rangle$ to TRL if $R_{dif} \neq \emptyset$ and Merge() if necessary
      RS-CVNN(TRL, $\langle p_i, R_{int}\rangle, p$) // see Algorithm 4
  else
    add $\langle p_i, R_i\rangle$ to TRL and Merge() if necessary
return TRL

hence we only need to examine the dominance relationship according to Lemma 1. RS-CVNN distinguishes four cases: (i) If p does not dominate *any interval* over R_{int}, i.e., $p \notin VC(l) \cup VC(r)$, the original tuple $\langle p_i, R_{int}\rangle$ remains valid and is added to TRL (line 2). (ii) If p dominates *entire* R_{int}, the algorithm replaces p_i with p and inserts $\langle p, R_{int}\rangle$ into TRL (lines 3-4). (iii) If p is only within the vicinity circle of l, i.e., p dominates *partial interval* on R_{int}, the algorithm calculates the intersection s_1 between the region R_{int} and the perpendicular bisector of the line segment $[p_i, p]$ (i.e., $\perp (p_i, p)$), and inserts $\langle p, [l, s_1]\rangle$ and $\langle p_i, [s_1, r]\rangle$ into TRL (lines 5-7). (iv) Similar to case (iii), if p is only inside the vicinity circle of r, the algorithm derives the intersection s_2 between R_{int} and $\perp (p_i, p)$, and adds $\langle p_i, [l, s_2]\rangle$ and $\langle p, [s_2, r]\rangle$ to TRL (lines 8-10).

Algorithm 4 Region Split for CVNN (RS-CVNN)

Input: a temporary result list TRL; a tuple $\langle p_i, R_{int}\rangle \in RL$ with region $R_{int} = [l, r]$; a data point p
/* $VC(p')$ denotes the vicinity circle of p', centered at p' with $dist(p, p')$ as radius */

1: **if** $p \notin VC(l)$ and $p \notin VC(r)$ **then**
2:   insert $\langle p_i, R_{int}\rangle$ into TRL
3: **else if** $p \in VC(l)$ and $p \in VC(r)$ **then**
4:   insert $\langle p, R_{int}\rangle$ into TRL
5: **else if** $p \in VC(l)$ **then**
6:   $s_1 = R_{int} \cap \perp (p_i, p)$
7:   insert both $\langle p, [l, s_1]\rangle$ and $\langle p_i, [s_1, r]\rangle$ into TRL
8: **else** // $p \in VC(r)$
9:   $s_2 = R_{int} \cap \perp (p_i, p)$
10:   insert both $\langle p_i, [l, s_2]\rangle$ and $\langle p, [s_2, r]\rangle$ into TRL

Figure 12 depicts an example with $P = \{a, b, c\}$, $O = \{o_1, o_2, o_3\}$ and $q = [s, e]$. Suppose point a has been processed and current $RL = \{\langle a, [s, s_2]\rangle, \langle\emptyset, [s_2, e]\rangle\}$. Now we invoke RLU to evaluate a new point b, with $VR_b = \{[s, s_3]\}$. RLU recursively checks each region in RL. First, $[s, s_2]$ is evaluated. As it overlaps with VR_b, RLU derives R_{int} ($= [s, s_2] \cap [s, s_3] = [s, s_2]$) and R_{dif} ($= [s, s_2] - [s, s_2] = \emptyset$), and calls RS-CVNN to examine whether a, the current VNN of R_{int}, can be partially/completely replaced by b. Since b is within the vicinity circle of s, RS-CVNN computes the intersection s_1 between $[s, s_2]$ and $\perp (a, b)$, i.e., the perpendicular bisector of the line segment $[a, b]$, and adds $\langle b, [s, s_1]\rangle$ and $\langle a, [s_1, s_2]\rangle$ to TRL. Next, RLU examines the second region in RL (i.e., $[s_2, e]$) and discovers it also overlaps with VR_b. Consequently, both R_{int} ($= [s_2, e] \cap [s, s_3] = [s_2, s_3]$) and R_{dif} ($= [s_2, e] - [s_2, s_3] = [s_3, e]$) are calculated. As the current VNN of $[s_2, e]$ is $\emptyset$, RLU adds $\langle b, [s_2, s_3]\rangle$ and $\langle\emptyset, [s_3, e]\rangle$ to TRL. Finally, it returns $TRL = \{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle b, [s_2, s_3]\rangle, \langle\emptyset, [s_3, e]\rangle\}$ as the updated result list RL.

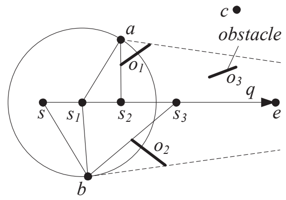
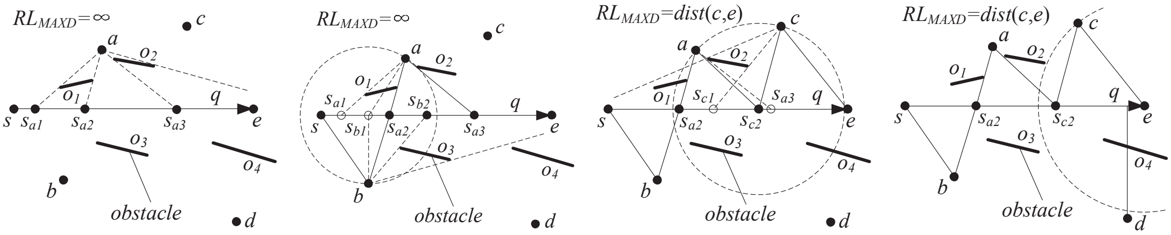


Fig. 12 Example of RLU algorithm

5.4 The complete CVNN search algorithm

Having explained GetObs, VRC, and RLU algorithms, we are ready to present the complete CVNN query processing algorithm, namely *CVNN Search Algorithm* (CVNN), whose pseudo-code is shown in Algorithm 5. CVNN takes as input an R-tree T_p on data set P, an R-tree T_o on obstacle set O, and a query line segment q, and outputs the final result list RL of a CVNN query. It follows the *best-first* traversal paradigm, as suggested by Heuristic 4.

In order to enable the best-first traversal, the algorithm maintains two heaps H_p and H_o to store the data and obstacle entries visited so far respectively, sorted by ascending order of their minimal distances (i.e., mindist) to q. First of all, CVNN enheaps the root nodes of T_p and T_o to H_p and H_o, respectively (line 2). Thereafter, it continuously deheaps the head entry e out of H_p for examination until H_p becomes empty (lines 3-14). Each examination involves two tasks. First, CVNN checks whether the early termination condition is satisfied, i.e., mindist$(e, q) \geq$ RL$_{\mathsf{MAXD}}$ (line 5). If yes, the algorithm terminates because the remaining entries in H_p can not contain any answer point according to Heuristic 1. Second, the entry e is evaluated. If e is a data point, CVNN invokes GetObs algorithm to obtain all the obstacles that may affect the visibility of e, calls VRC algorithm to derive e's visible region VR_e over q, and utilizes the RLU algorithm to update the current result list RL (lines 7-10). Otherwise, e must refer to an intermediate node (i.e., a non-leaf entry). CVNN visits its subtree only if it may contain any



(a) After processing a (b) After processing b (c) After processing c (d) After processing d

Fig. 13 Illustration of a CVNN query processing

qualifying data point via Heuristic 2 and Heuristic 3 (lines 11-14). The advantage of CVNN algorithm over exhaustive scan is that the access to some unnecessary nodes, i.e., those certainly not containing any qualified object, is eliminated.

Algorithm 5 CVNN Search (CVNN)

Input: a data R-tree T_p; an obstacle R-tree T_o; a query line segment $q = [s, e]$
Output: the result list RL of a CVNN query
/* $T_p.root$ denotes the root node of T_p; $T_o.root$ represents the root node of T_o */

```
1:  RL = {⟨∅, [s, e]⟩}, RL_MAXD = ∞, and L_o = ∅
2:  H_p = {T_p.root}, H_o = {T_o.root}
3:  while H_p ≠ ∅ do
4:      de-heap the top entry e of H_p
5:      if mindist(e, q) ≥ RL_MAXD then      // use Heuristic 1
6:          break
7:      else if e is a data point then
8:          GetObs(T_o, H_o, mindist(e, q), q, L_o)   // See Algorithm 1
9:          VR_e = VRC(e, q, L_o)        // See Algorithm 2
10:         RL = RLU(RL, e, VR_e)        // See Algorithm 3
11:     else
12:         for each child entry e_i ∈ e do
13:             if e_i dominates a subinterval of any region in RL and
                it is visible to q then     // use Heuristics 2 and 3
14:                 insert e_i into H_p
15: return RL
```

Consider the example depicted in Figure 13, where $P = \{a, b, c, d\}$, $O = \{o_1, o_2, o_3, o_4\}$, and $q = [s, e]$. Initially, the result list RL is set to $\{\langle\emptyset, [s, e]\rangle\}$. When the first data point a (that is the closest to q without considering obstacles) is visited, CVNN invokes GetObs to obtain all the obstacles that may affect the visibility of a (i.e., o_1 and o_2). Then, it uses VRC to get $VR_a = \{[s, s_{a_1}], [s_{a_2}, s_{a_3}]\}$, i.e., the a's visible regions over q. Next, RLU is called to update the current RL to $\{\langle a, [s, s_{a_1}]\rangle, \langle\emptyset, [s_{a_1}, s_{a_2}]\rangle, \langle a, [s_{a_2}, s_{a_3}]\rangle, \langle\emptyset, [s_{a_3}, e]\rangle\}$, as shown in Figure 13(a). The second point examined is b. Since b dominates $[s, s_{a_1}]$ and $[s_{a_1}, s_{a_2}]$, the corresponding VNNs are replaced by b with $RL = \{\langle b, [s, s_{a_2}]\rangle, \langle a, [s_{a_2}, s_{a_3}]\rangle, \langle\emptyset, [s_{a_3}, e]\rangle\}$, as shown in Figure 13(b). Subsequently, CVNN evaluates the third point c and updates RL to $\{\langle b, [s, s_{a_2}]\rangle, \langle a, [s_{a_2}, s_{c_2}]\rangle, \langle c, [s_{c_2}, e]\rangle\}$, which is illustrated in Figure 13(c). Finally, when the last point d is encountered, d is pruned directly as $\mathsf{mindist}(d, q) > \mathsf{RL_{MAXD}}$ $(= dist(c, e))$. Here, the algorithm terminates with the final query result $RL = \{\langle b, [s, s_{a_2}]\rangle, \langle a, [s_{a_2}, s_{c_2}]\rangle, \langle c, [s_{c_2}, e]\rangle\}$, as shown in Figure 13(d).

Next, we reveal some characteristics of the CVNN algorithm, analyze its time complexity, and prove its correctness.

Lemma 4 *Every data point in a data set P will be examined during the* CVNN *search, unless one of its ancestor nodes has been pruned.*

Proof The proof is obvious since all data points in P that are not pruned by Heuristics 1 to 4 (proposed in Section 4.1) are inserted into the heap and examined. □

Lemma 5 *Only obstacles that may impact the visibility of the current data point processed are maintained in L_o.*

Proof The proof is straightforward because the CVNN algorithm employs the GetObs algorithm to find incrementally all the obstacles that might affect the visibility of the data point processed currently, and enables heuristics 5 to 7 (presented in Section 4.2) to prune away all the non-qualifying obstacles that cannot contribute to the final query result. □

Lemma 6 *The* CVNN *algorithm traverses the data R-tree T_p and the obstacle R-tree T_o* at most once.

Proof As shown in Algorithm 5, the CVNN algorithm traverses T_p *once* based on the best-first manner to evaluate every data point in P that cannot be pruned. In addition, it only traverses T_o *once*. Although the GetObs algorithm is invoked every time a new data point $p \in P$ is evaluated, it utilizes the obstacles preserved in the current L_o and traverses the T_o in an incremental fashion. □

Let $|O|$ be cardinality of the obstacle set O, $|P|$ be the cardinality of the data set P, $|\mathsf{IRC}|$ be the time complexity of the IRC function called by VRC, $|IR_{p,o}|$ be the maximum number of regions in $IR_{p,o}$ (used in VRC), and $|RL|$ be the maximum cardinality, in terms of number of tuples, of a result list RL. The time complexity of CVNN algorithm is presented in Theorem 1, while its correctness is proved in Theorem 2.

Theorem 1 *The time complexity of the* CVNN *algorithm is* $O\big(|P|\cdot\log|P|\cdot(|O|\cdot\log|O|+|O|\cdot(|\mathsf{IRC}|+|IR_{p,o}|)+|RL|)\big)$.

Proof As mentioned before, CVNN actually invokes GetObs, VRC, and RLU to evaluate each point, and hence its time complexity is attributed to by that of GetObs, VRC, and RLU. First, all the obstacles in O have to be accessed in the worst case, and thus the time complexity of the GetObs algorithm is $O(|O| \cdot \log |O|)$. Next, VRC is to obtain the visible region of the point processed currently over q by considering each obstacle preserved in L_o. Consequently, its time complexity is $O(|O| \cdot (|\mathsf{IRC}| + |IR_{p,o}|))$. Third, RLU has to check every tuple in RL, and the RS-CVNN invoked by RLU can be completed in $O(1)$ time. Consequently, the time complexity of RLU is $O(|RL|)$. Given the fact that CVNN needs to evaluate all the points in P in the worst case and it takes $O(\log |P|)$ to locate a point in P, the overall time complexity of the CVNN algorithm is $O\big(|P| \cdot \log |P| \cdot (|O| \cdot \log |O| + |O| \cdot (|\mathsf{IRC}| + |IR_{p,o}|) + |RL|)\big)$. □

Theorem 2 *The* CVNN *algorithm retrieves* exactly *the VNN of every point along a given query line segment, i.e., the algorithm has no* false misses *and no* false hits.

Proof First, no answer points are missed (i.e., no false negatives) because only unqualified data points in P are pruned away safely according to Heuristics 1 through 4. Second, the impact of each qualified data point in P on the current result list RL is evaluated, which ensures no false positives (i.e., no false hits). □

5.5 CVNN query processing on one R-tree

Our previously presented CVNN search algorithm assumes that dataset P and obstacle set O are indexed by two *separate* R-trees. In what follows, we explain how to extend it to support CVNN search on a *single* R-tree that indexes both data points and obstacles.

The detailed extensions are listed as follows: (i) It requires only one heap H to store candidate entries (containing data points, obstacles, and non-leaf nodes), sorted in ascending order of their minimum distances to a given query line segment q. (ii) When processing the top entry e removed from H, it distinguishes three cases. (1) e is an obstacle. It adds e to a linked list L_o, which maintains all the obstacles that may affect the visibility of the data points processed so far with respect to q. (2) e is a data point p. It computes the visible region of e over q, and updates the current result list RL if necessary. According to the Heuristic 7, any obstacle o that may impact the visibility of e on q must satisfy the condition: $\mathsf{mindist}(o, q) < \mathsf{mindist}(e, q)$. Therefore, all the obstacles that might affect the dominance of p must have been visited before p. Note that, there is no need to invoke the GetObs algorithm to get all the obstacles that may affect e's visibility, since both data points and obstacles are indexed by one unified R-tree. (3) e is a non-leaf node, indicating that it may contain data points and/or obstacles. The subtrees of e that may contain answer points or obstacles that might affect the visibility of some answer points are retrieved. Note that all the heuristics proposed in Section 4 can still be applied for pruning unnecessary node accesses significantly.

6 Variations of CVNN queries

In this section, we study several interesting CVNN variants, and present how the proposed CVNN algorithm can be adapted accordingly. In particular, four variants are defined, including (i) continuous visible kNN (CVkNN) search, (ii) trajectory VNN (TVNN) retrieval, (iii) CVNN query with visible distance threshold δ (δ-CVNN), and (iv) constrained CVNN (CCVNN) search. It is important to note that due to the space limitation and the similarity of the algorithm extensions, we only explain the extension of the algorithm to support CVkNN retrieval in detail.

6.1 CVkNN search

Given a data set P, an obstacle set O, and a query line segment $q = [s, e]$, CVkNN search is to retrieve k VNNs for every point on q. The result list RL of a CVkNN query contains a set of $\langle S, R\rangle$ tuples, where S is the set of VNNs for all the points along the region/interval $R \subseteq q$. Different from conventional kNN retrieval, the answer set S might not exist (i.e., $S = \emptyset$) or it might not hold k answer points (i.e., $|S| < k$), due to the existence of obstacles. The proposed algorithms for CVNN queries can be easily extended to support CVkNN search. The detailed extensions are described as follows.

First, the VNN query defined in Definition 3 is replaced by a general VkNN query, stated in Definition 8. Accordingly, the CVkNN query is defined in Definition 9.

Definition 8 Visible k nearest neighbors [4]. Given a query point q', $p \in P$ is one of the *visible k nearest neighbors* (VkNNs) of q' iff: (i) p is visible to q'; and (ii) there are at most $(k-1)$ data points $p' \in P - \{p\}$ such that p' is visible to q' and meanwhile has its distance to q' smaller than that from p to q', i.e., $|\{p' \in P - \{p\} | p'$ is visible to $q' \wedge dist(p, q') > dist(p', q')\}| < k$.

Definition 9 Continuous visible k nearest neighbor query. Given P, O, and q, a *continuous visible k nearest neighbor* (CVkNN) *query* returns a result list RL that contains a set of $\langle S_i, R_i\rangle$ $(i \in [1, t])$ tuples such that (i) $\forall i, j \in [1, t] (i \neq j)$, $R_i \cap R_j = \emptyset$[10]; (ii) $\cup_{i=1}^{t} R_i = q$; (iii) $\forall i \in [1, t]$, $|S_i| \leq k$; and (iv) $\forall \langle S_i, R_i\rangle \in RL$, if $S_i \neq \emptyset$, S_i is the set of VkNNs of all points along R_i.

[10] As with Definition 4, if R_i and R_j are *adjacent*, i.e., $|i - j| = 1$, $R_i \cap R_j \neq \emptyset$.

Second, Heuristic 1 requires updating. $\mathsf{RL_{MAXD}}$ is replaced with $MAX_{1\leq i\leq|RL|}(\mathsf{maximumdist}(S_i, R_i.l), \mathsf{maximumdist}\ (S_i, R_i.r))$, where $|RL|$ denotes the number of regions in the current result list RL, and set S_i maintains the set of VkNNs retrieved so far for its corresponding region R_i. The detailed proof is presented in Lemma 7.

Lemma 7 *Assume the set S contains the VkNN objects identified so far for the region R. A new point $p \in P$ updates S iff $dist(p, R.l) <$ $\mathsf{maximumdist}(S, R.l)$ or/and $dist(p, R.r) <$ $\mathsf{maximumdist}(S, R.r)$, with $\mathsf{maximumdist}(S, r)$ defined as follows:*

$$\mathsf{maximumdist}(S, r) = \begin{cases} MAX_{\forall p_i \in S} dist(p_i, r) & if\ |S| = k \\ \infty & if\ |S| < k \end{cases}$$

Proof When $|S| < k$, p will be included in S based on Definition 8. Now we would like to prove the case when $|S| = k$. First, we proof sufficiency. Without loss of generality, we assume $dist(p, R.l) <$ $\mathsf{maximumdist}(S, R.l)$ and $\exists p_i \in S$ such that $dist(p, R.l) < dist(p_i, R.l)$. According to Lemma 1 (proposed in Section 3.2), it is guaranteed that p violates the dominance of p_i over R. Given the fact that $|S - \{p_i\}| = (k-1)$ and Definition 8, p_i is no longer one of the VkNNs of $R.l$, and S needs to update. Next, we proof necessity. Suppose p updates R by replacing $p_i \in S$. Hence, there is at least one point $r \in R$ such that $dist(r, p_i) > dist(r, p)$. As demonstrated in Lemma 1, p must be within the vicinity circle $cir(r, p_i)$ and thus the union of $cir(R.l, p_i)$ and $cir(R.r, p_i)$. In other words, $dist(p, R.l) < dist(p_i, R.l) \leq$ $\mathsf{maximumdist}(S, R.l)$ or/and $dist(p, R.r) < dist(p_i, R.r)$ $\leq \mathsf{maximumdist}(S, R.r)$. The proof completes. □

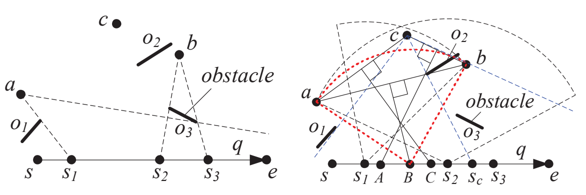


Fig. 14 Illustration of updating RL for a CV2NN query

The evaluation of data points is similar to that for CVNN search. Specifically, the evaluation of each data point $p \in P$ involves three steps. First, all the obstacles affecting p's visibility are obtained. Second, p's visible region VR_p over q is derived. Third, the current result list RL is updated if necessary, which is more complex than that under CVNN (i.e., $k = 1$) retrieval. We use an example depicted in Figure 14 to illustrate the update operation. Suppose a CV2NN ($k = 2$) query is issued with data set $P = \{a, b, c\}$, obstacle set $O = \{o_1, o_2, o_3\}$, and query line segment $q = [s, e]$.

We assume points a, b have been processed, and currently $RL = \{\langle\{b\}, [s, s_1]\rangle, \langle\{a, b\}, [s_1, s_2]\rangle, \langle\{a\}, [s_2, s_3]\rangle, \langle\{a, b\}, [s_3, e]\rangle\}$, as shown in Figure 14(a). Notice that the number of the current VNN(s) for intervals $[s, s_1]$ and $[s_2, s_3]$ is only one due to the obstruction of obstacles. Now the evaluation of a new data point c starts, and assume that we have got its visible region $VR_c = \{[s, s_c]\}$ on q.

To simplify our discussion, we only focus on the evaluation of c based on a specified interval R, but the same process can be applied to other intervals in RL. First, according to the visibility, c partitions the interval R into two regions R_{int} and R_{dif}, with $R_{int} = R \cap VR_c$ and $R_{dif} = R - R_{int}$. Point c might change the result corresponding to R_{int}, but definitely not R_{dif}. Consequently, the evaluation can be safely terminated if $R_{int} = \emptyset$, meaning that any point on the interval R is *invisible* to c. Now suppose $R_{int} = [l, r] \neq \emptyset$, and its corresponding answer point set is S. If $|S| < k$, c becomes an answer point for every point on R_{int}. As an example, consider the evaluation of c over region R (= $[s_2, s_3]$) $\subseteq q$ (= $[s, e]$). Since $R \cap VR_c$ = $[s_2, s_c]$, the tuple $\langle\{a\}, [s_2, s_3]\rangle$ is changed to $\langle\{a, c\}, [s_2, s_c]\rangle$ and $\langle\{a\}, [s_c, s_3]\rangle$. If $|S| = k$, We have to check whether (i) $\mathsf{maximumdist}(S, l) > dist(c, l)$ and/or (ii) $\mathsf{maximumdist}(S, r) > dist(c, r)$ hold. If neither condition is satisfied, c is discarded as it cannot be an answer point to any point along R. Otherwise, the interval R_{int} needs to be split, which is tackled by the RS-CVkNN algorithm presented in Algorithm 6.

RS-CVkNN evaluates the impact of a new data point p on an interval $R_{int} \subseteq R$. It takes as inputs a temporary result list TRL, a region $R_{int} = [l, r]$, a result set S that keeps k answer points for R_{int} identified so far, and a new data point p, and returns the updated TRL. RS-CVkNN distinguishes two cases: (i) p is not an answer point of any point along R_{int}, i.e., $dist(p, l) > \mathsf{maximumdist}(S, l)$ and $dist(p, r) > \mathsf{maximumdist}(S, r)$. In this case, p certainly will not change S and $\langle S, R_{int}\rangle$ remains valid (lines 1-2). (ii) p is an answer point of some points on R_{int} , i.e., $dist(p, l) \leq \mathsf{maximumdist}\ (S, l)$ and/or $dist(p, r) \leq \mathsf{maximumdist}(S, r)$. In this case, the algorithm performs the following tasks. First, for every point $p_i \in S$, the intersection between R_{int} and $\perp (p_i, p)$ is computed and inserted into a temporary set S_{sp} (lines 4-6). Then, it finds the point sp ($\neq l$) in S_{sp} that is the closest to the starting point of R_{int} (i.e., l) (line 7). If sp does not exist, the tuple $\langle S, R_{int}\rangle$ remains valid and is added to TRL (lines 8-9). Otherwise, RS-CVkNN locates point $p' \in S$ that generates sp, i.e., $sp = R_{int} \cap \perp (p', p)$, and point $p'' \in S$ that has the maximal distance to l, i.e., $dist(p'', l) = \mathsf{maximumdist}(S, l)$ (line 11). If p is closer to l than p'', the algorithm first swaps the values of p and p'', and then RS-CVkNN is invoked recursively to check the validity of S on region $[l, r]$ upon the existence of p (lines 12-14). Here, note that set S is changed as $p'' \in S$ changes its value, and the evaluated point p is updated as well. Otherwise, p'' is still closer to l than p. The algorithm maintains the tuple $\langle S, [l, sp]\rangle$ in TRL and calls RS-CVkNN again to examine the validity of $S - \{p'\} \cup \{p\}$ on region $[sp, r]$ upon the existence of p' (lines 15-17). Finally, the new result list TRL is returned to complete the algorithm.

Algorithm 6 Region Split for CVkNN (RS-CVkNN)

Input: a temporary result list TRL; a region $R_{int} = [l, r]$; a set S of the current VkNNs for each point along R_{int}; a data point p
Output: the updated TRL

1: **if** $dist(p, l) >$ maximumdist(S, l) and $dist(p, r) >$ maximumdist (S, r) **then** // p does not dominate R_{int}
2: insert $\langle S, R_{int}\rangle$ into TRL
3: **else** // p dominates R_{int} partially/completely
4: **for** each point $p_i \in S$ **do**
5: $sp_i = R_{int} \cap \perp (p_i, p)$
6: add sp_i to set S_{sp}
7: let $sp(\neq l)$ be the point in S_{sp} that is the closest to l
8: **if** sp does not exist **then**
9: insert $\langle S, R_{int}\rangle$ into TRL
10: **else**
11: let p' be the point in S such that $sp = R_{int} \cap \perp (p', p)$, and p'' be the point in S satisfying $dist(p'', l) =$ maximumdist(S, l)
12: **if** $dist(p, l) < dist(p'', l)$ **then**
13: swap(p, p'')
14: RS-CVkNN$(TRL, [l, r], S, p)$ // $p = p''$
15: **else**
16: insert $\langle S, [l, sp]\rangle$ into TRL
17: RS-CVkNN$(TRL, [sp, r], S - \{p'\} \cup \{p\}, p')$
18: return TRL

Back to the running example shown in Figure 14. Consider the evaluation of c over region $[s_1, s_2] \subseteq q$ (= $[s, e]$), whose corresponding answer point set S is $\{a, b\}$. Point c is fully visible to $[s_1, s_2]$, and $dist(c, s_1) <$ maximumdist(S, s_1) $(= dist(b, s_1))$ and $dist(c, s_2) <$ maximumdist(S, s_2) $(= dist\ (a, s_2))$. Therefore, RS-CVkNN is employed to find the split points along $[s_1, s_2]$. At the first recursion, the intersection A between q and $\perp\ (b, c)$ as well as the intersection C between q and $\perp\ (a, c)$ are derived. Thus, S_{sp} = $\{A, C\}$, and sp be point A as it is closer to s_1. Accordingly, we locate b as p' whose bisector contributes to the generation of A and b as p'' with the maximal distance to s_1. Since $dist(c, s_1) < dist(b, s_1)$, the algorithm understands the c will replace b to become V2NN objects to s_1, together with a. Thereafter, RS-CVkNN (TRL, $[s_1, s_2]$, $\{a, c\}$, b) is called again to evaluate the impact of b on $[s_1, s_2]$ with $S = \{a, c\}$.

Again, S_{sp} = $\{A, B\}$ is formed first with A contributed by $\perp\ (b, c)$ and B contributed by $\perp\ (a, b)$, as illustrated in Figure 14(b). Given S_{sp}, the one closest to s_1, i.e., A, is located, and p' and p'' are both set to c. As c is closer to s_1 than b does, $S = \{a, c\}$ remains valid for $[s_1, A]$, and $\langle\{a, c\}, [s_1, A]\rangle$ is inserted into TRL. Next, RS-CVkNN (TRL, $[A, s_2]$, $\{a, b\}$, c) is invoked with the query line segment shrink to $[A, s_2]$. The algorithm proceeds in the same manner until all the split points along the interval $[s_1, s_2]$ are found, after which TRL is updated to $\{\langle\{a, c\}, [s_1, A]\rangle, \langle\{a, b\}, [A, C]\rangle, \langle\{b, c\}, [C, s_2]\rangle\}$. Table 2 lists the executive processes of RS-CVkNN.

It is worth mentioning that k has a direct impact on the size of the result list RL. In particular, the greater the k is, the larger the number of regions contained in RL is, and the higher the cost incurred by CVkNN algorithm is.

6.2 Trajectory VNN search

The above CVNN search is on a *single* query line segment. However, in real applications, users may want to retrieve the VNN of every point along a specified *trajectory* that consists of several consecutive line segments. For example, a wildlife observer in Yellow Stone National Park may issue a query to find the closest observation point where he/she is most likely to see wolves along his/her hiking trail. Motivated by this, we introduce trajectory VNN (TVNN) search, which retrieves the VNN of every point along a given query trajectory, and the proposed CVNN algorithm can be adapted to handle TVNN retrieval as well.

An intuitive solution to the TVNN query, namely *Simple Processing Algorithm* (SP), is to invoke the CVNN algorithm for each line segment q_i included in the trajectory q (i.e., $\forall q_i \subseteq q$), to find the VNN of every point along q_i; and then merge the results if necessary. Although this approach is straightforward, it is inefficient in terms of I/O cost, which will be demonstrated by our experimental results to be presented in Section 7.4. This is because, given a query trajectory q that contains $|q|$ line segments (i.e., $q = \cup_{1 \leq i \leq |q|} q_i$), SP needs to traverse the data R-tree T_p and the obstacle R-tree T_o $|q|$ times, resulting in *extremely high* I/O overhead, especially when $|q|$ is large. In the sequel, we explain how to extend the CVNN algorithm to tackle the TVNN query by traversing T_p and T_o *only once*.

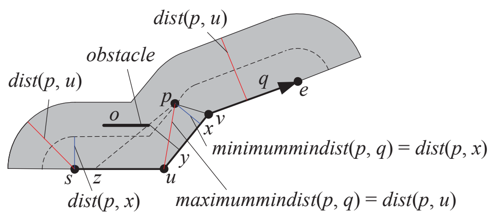


Fig. 15 Distance metrics of TVNN search

First, instead of decomposing the trajectory into multiple line segments, we consider it as one unit. The minimal distance between an entry E (representing a data point or an obstacle or a node MBR) and a specified query trajectory q is defined as the *minimum* distance among all the shortest distances from E to each line segment $q_i \subseteq q$, i.e.,

Table 2 The trace of RS-CVkNN algorithm

# **Recursion**	S_{sp}	sp	p'	p''	**Operation**	TRL
First recursion	$\{A, C\}$	A	b	b	swap(c, b), RS-CVkNN(TRL, $[s_1, s_2]$, $\{a, c\}$, b)	$\emptyset$
Second recursion	$\{A, B\}$	A	c	c	insert $\langle\{a, c\}, [s_1, A]\rangle$ into TRL, RS-CVkNN(TRL, $[A, s_2]$, $\{a, b\}$, c)	$\{\langle\{a, c\}, [s_1, A]\rangle\}$
Third recursion	$\{A, C\}$	C	a	b	insert $\langle\{a, b\}, [A, C]\rangle$ into TRL, RS-CVkNN(TRL, $[C, s_2]$, $\{b, c\}$, a)	$\{\langle\{a, c\}, [s_1, A]\rangle, \langle\{a, b\}, [A, C]\rangle\}$
Fourth recursion	$\{B, C\}$	$-$	$-$	c	insert $\langle\{b, c\}, [C, s_2]\rangle$ into TRL,	$\{\langle\{a, c\}, [s_1, A]\rangle, \langle\{a, b\}, [A, C]\rangle, \langle\{b, c\}, [C, s_2]\rangle\}$

minimummindist$(E, q) = MIN_{1 \leq i \leq |q|}$(mindist$(E, q_i)$). In Figure 15, for example, minimummindist $(p, q) = MIN($ mindist$(p, [s, u])$, mindist$(p, [u, v])$, mindist $(p, [v, e])) = MIN(dist(p, u), dist(p, x), dist(p, v)) = dist(p, x)$.

Second, given a query line segment q_i of a query trajectory q, the obstacles o that might affect the visibility of point p over q_i must have their minimal distances to q_i bounded by mindist(p, q_i). As we need to retrieve all the obstacles that might affect p's visibility on any point along q, the retrieval distance r requested by GetObs algorithm should be set to the *maximum* distance of all the minimal distances from p to every line segment $q_i \subseteq q$, i.e., maximummindist(p, q) $= MAX_{i \in [1, |q|]}$ (mindist(E, q_i)). For example, as shown in Figure 15, maximummindist$(p, q) = MAX($ mindist$(p, [s, u])$, mindist$(p, [u, v])$, mindist$(p, [v, e])) = dist\ (p, u)$, and the shaded area covers all the obstacles affecting p's visibility over q.

Third, the pruning heuristics proposed for CVNN search are still applicable, but based on the above minimummindist and maximummindist metrics.

Compared with SP, this approach processes a TVNN search via a *single* traversal of the R-trees no matter how many line segments the specified query trajectory contains, and thus, it has lower I/O cost. Compared with SP, however, the method has higher CPU overhead because it incurs more distance computation, visibility check, and result list update, which will also be demonstrated by our experimental results to be presented in Section 7.4.

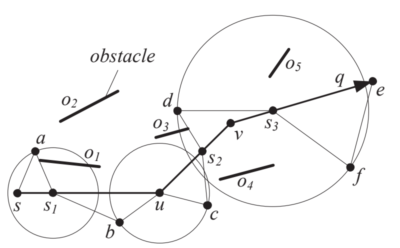


Fig. 16 Example of a TVNN query

Figure 16 shows an example, where a data set $P = \{a, b, c, d, f\}$, an obstacle set $O = \{o_1, o_2, o_3, o_4, o_5\}$, and a query trajectory $q = [s, e]$ consisting of 3 line segments, i.e., $q_1 = [s, u]$, $q_2 = [u, v]$, and $q_3 = [v, e]$. As depicted in Figure 16, the final result of a TVNN query is $\{\langle a, [s, s_1]\rangle, \langle b, [s_1, u]\rangle, \langle c, [u, s_2]\rangle, \langle d, [s_2, s_3]\rangle, \langle f, [s_3, e]\rangle\}$, after processing points b, c, d, a, f (in this order).

Note that the complexity of TVNN retrieval, compared to the CVNN query, is higher due to the fact that the number of split points and the number of obstacles which need to be considered increase with the number of query line segments.

6.3 CVNN query with distance threshold δ

In real life, users might want to enforce distance constraints on CVNN queries. Take the application *placement of traffic surveillance cameras* described in Section 1 as an example. As different cameras have various crop factors, a camera has a limited imaging area. Suppose cameras installed can only monitor the objects located within 10 meters, CVNN search with *distance threshold* 10 is more suitable, compared with conventional CVNN retrieval. In view of this, we introduce δ-CVNN search, a CVNN retrieval with *maximum visible distance δ constraint*. Given a data set P, an obstacle set O, a query line segment q, and a distance threshold δ, a δ-CVNN query returns the VNN of every point along q with its distance to q bounded by δ.

A straightforward approach to answer δ-CVNN query is to first perform CVNN search, and then filter out those answer objects whose distances to the corresponding intervals on a specified query line segment q exceed δ. Nevertheless, this method is not very efficient since the distance constraint δ is applied at a very late stage. Another solution is to retrieve all the objects with their distances to q not exceeding δ and then conduct CVNN retrieval. However, it needs to build a new R-tree on these objects. In fact, the proposed algorithms for CVNN search can be easily adjusted to support δ-CVNN retrieval, by integrating distance threshold δ during the query processing. Moreover, in addition to all the Heuristics presented in Section 4 that are still applicable, we also develop two new pruning heuristics to fully utilize the distance constraint δ in order to further improve the search performance.

Search early termination. As the tree built on the data set P is traversed in a best-first manner, the algorithm can be safely terminated once an entry E (data point or node

MBR) with mindist to q larger than δ is encountered. This is because, when the top entry E has its mindist to q exceeding δ, it is guaranteed that all the unexamined data points have their distances to q greater than δ and thus will be excluded from the final query result.

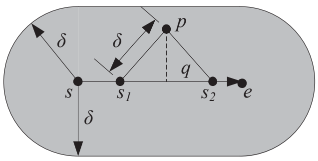


Fig. 17 Search range of δ-CVNN

Search range shrinking. The search space of δ-CVNN retrieval is limited by δ. As an example, the shadowed area in Figure 17 represents the search space of a δ-CVNN query with $q = [s, e]$ and δ being the distance threshold. Consequently, any entry (involving data point, obstacle, and node MBR) that is outside the search space can be directly excluded from the further examination, since it cannot contribute to the final query result.

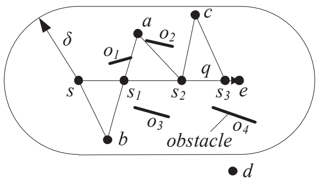


Fig. 18 Example of a δ-CVNN query

As shown in Figure 18, an example δ-CVNN query is issued at $q = [s, e]$, with $P = \{a, b, c, d\}$ and $O = \{o_1, o_2, o_3, o_4\}$. The δ-CVNN search outputs $\{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle c, [s_2, s_3]\rangle, \langle \emptyset, [s_3, e]\rangle\}$, which is different from the output $\{\langle b, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle c, [s_2, e]\rangle\}$ of a CVNN query issued at q. Take the interval $[s_3, e]$ as an example, its VNN c is an answer object to CVNN query but not δ-CVNN as its distance to its dominance region $[s_3, e]$ exceeds threshold δ.

6.4 Constrained CVNN search

Our previously proposed CVNN search and its variants (including CVkNN, TVNN, and δ-CVNN queries) aim at finding, from the *entire* data space, the VNN for each point along a given query line segment (or a query trajectory) q. However, in some real applications, users might be *only* interested in the objects within a specified spatial region. Take the application *tourist recommendation* presented in Section 1 as an example. Suppose a photographer wants to take photography of scenes in the sunset. As only objects within a limited range of distances from the camera will be reproduced clearly, the photographer may be only interested in the nearby visible scenes within a certain distance to his planed traveling route. Motivated by this, we introduce a CVNN query with a spatial region constraint, namely *constrained CVNN (CCVNN) search*. Given a data set P, an obstacle set O, a query line segment q, and a constrained region R_c, a CCVNN query retrieves, for every point along q, the VNN among all the objects located inside R_c. Actually, this type of queries is to apply conventional (i.e., unconstrained) CVNN retrieval in a specified spatial region and thus the final result of CCVNN search must satisfy the given region constraints.

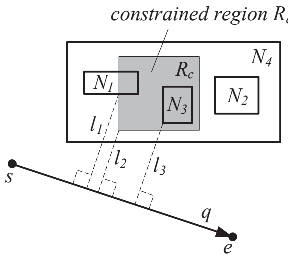
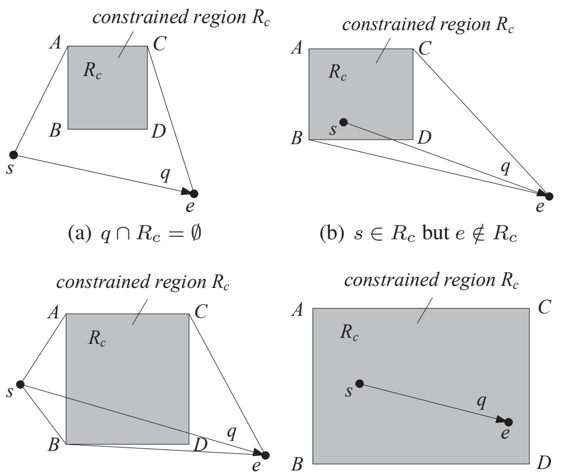


Fig. 19 Illustration of $\mathsf{mindist}(E, q, R_c)$

Since the users are only interested in the objects/points located inside a specified spatial region, a simple method for answering CCVNN search is to first find out all the data points that are within the given spatial region, denoted as $P_{R_c} = \{p \in P \wedge p \in R_c\}$, and then perform a CVNN query based on P_{R_c} and the obstacle set O. However, this approach requires to construct an R-tree on P_{R_c} during the query processing dynamically. Alternatively, we extend the proposed CVNN algorithm to handle CCVNN retrieval, by integrating the examination of regional constraint conditions during the search. In the following, we highlight two main differences between CVNN search and CCVNN search.

First, conventional (i.e. unconstrained) CVNN search traverses the dataset P in a best-first fashion as long as the entry E (data point or node MBR) has $\mathsf{mindist}(E, q)$ bounded by the current $\mathsf{RL_{MAXD}}$. Nevertheless, CCVNN retrieval only visits those entries E that overlap with R_c according to ascending order of $\mathsf{mindist}(E, q, R_c)$. Here, $\mathsf{mindist}(E, q, R_c) = \mathsf{mindist}(E \cap R_c, q)$. As illustrated in Figure 19, $\mathsf{mindist}(N_1, q, R_c) = l_1$, $\mathsf{mindist}(N_2, q, R_c) = \infty$, $\mathsf{mindist}(N_3, q, R_c) = \mathsf{mindist}\,(N_3, q) = l_3$, and $\mathsf{mindist}(N_4, q, R_c) = \mathsf{mindist}(R_c, q) = l_2$. In addition, CCVNN search can terminate the search when the heap H_p becomes empty or $\mathsf{mindist}(E, q, R_c)$ of the top entry E in a heap H_p that contains unvisited node entries is larger than the current $\mathsf{RL_{MAXD}}$. Furthermore, Heuristics 1 to 4 (presented in Section 4.1) are also applicable except that, for each intermediate node entry E, $\mathsf{mindist}\,(E, q)$ is replaced with $\mathsf{mindist}(E, q, R_c)$.

Second, the search range for the obstacles that might affect the visibility of some candidate answer points is restricted by the region bounded by R_c and the query line segment q. According to the relative locations between q and R_c, there are four possible situations, as depicted in Figure 20.



(a) $q \cap R_c = \emptyset$ (b) $s \in R_c$ but $e \notin R_c$

(c) $s, e \notin R_c \wedge q \cap R_c \neq \emptyset$ (d) $q \subseteq R_c$

Fig. 20 Relative positions of a query line segment $q = [s, e]$ and a constrained region R_c

Specifically, (i) q and R_c are disjoint, as shown in Figure 20(a). CCVNN retrieval only needs to access those obstacles located inside the quadrilateral $AseC$. (ii) One of the endpoints of q falls into R_c, as illustrated in Figure 20(b). CCVNN search only needs to visit those obstacles located within the quadrilateral $ABeC$. (iii) q intersects R_c, as depicted in Figure 20(c). CCVNN query only needs to scan those obstacles located inside the pentagon $AsBeC$. (iv) q falls into R_c completely, as shown in Figure 20(d). CCVNN search only needs to access those obstacles located inside R_c. Consequently, when the GetObs algorithm is invoked to retrieve the obstacles affecting the visibility of a data point $p \in P$ processed currently, only the obstacles located inside the restricted search range require examination. In addition, Heuristics 5 to 7 can still be applied for obstacle pruning.

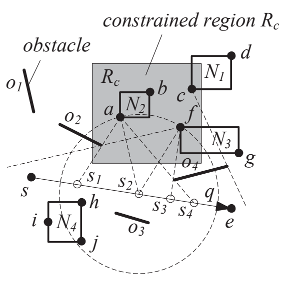


Fig. 21 Example of a CCVNN query

Consider the CCVNN query depicted in Figure 21 with $P = \{a, b, c, d, f, g, h, i, j\}$, $O = \{o_1, o_2, o_3, o_4\}$, $q = [s, e]$, and R_c set to the shaded rectangle. The final query result list $RL = \{\langle f, [s, s_1]\rangle, \langle a, [s_1, s_2]\rangle, \langle f, [s_2, s_3]\rangle, \langle a, [s_3, s_4]\rangle, \langle \emptyset, [s_4, e]\rangle\}$. Notice that although points g, h, i and j are closer to q than points a and f, they are excluded from the RL as they are outside R_c.

It is worth mentioning that the constrained region R_c has a significant impact on the algorithm performance, as demonstrated by our experimental results (to be presented in Section 7.6). Specifically, if a specified query line segment q is *far away* from R_c, more obstacles might affect the visibility of a data point inside R_c over q, which leads to more obstacle retrieval, visibility examination, and result list updating.

7 Performance evaluation

In this section, we evaluate the performance of the proposed pruning heuristics and CVNN search algorithm for CVNN query and its variants via extensive experiments. All the algorithms were implemented in C++ and the experiments were conducted on a Pentium IV 3.0 GHz PC with 2GB RAM, running Microsoft Windows XP Professional Edition. The detailed experimental setup is presented in Section 7.1. Five sets of experiments are conducted. The first set evaluates the effectiveness of the heuristics proposed, as reported in Section 7.2. The second set verifies the efficiency and effectiveness of CVNN algorithm in supporting CVkNN search with $k \geq 1$, presented in Section 7.3. The third, fourth, and fifth sets of experiments are to evaluate the flexibility of CVNN algorithm in supporting TVkNN, δ-CVkNN, and CCVkNN queries, different variants of CVNN queries, in Section 7.4, Section 7.5, and Section 7.6, respectively.

Table 3 Statistics of the real datasets used

Dataset	Size	Format	Description
LCA	62,556	2D points	locations in California
CGR	5,922	2D points	cities and villages in Greece
SLA	131,461	2D MBRs	streets in Los Angeles
RGR	24,650	2D MBRs	rivers in Greece

7.1 Experimental setup

Our experiments are based on both real datasets and synthetic datasets, with the search space fixed at a $[0, 10000] \times [0, 10000]$ square. Four real datasets are deployed, namely ***LCA***, ***CGR***, ***SLA***, and ***RGR***[11], as summarized in Table 3. Specifically, *LCA* and *CGR* contain two-dimensional (2D) points, representing $62,556$ locations in California and $5,922$

[11] *LCA*, *CGR*, *SLA*, and *RGR* datasets are available in the R-tree Portal (*http://www.rtreeportal.org*).

Table 4 Parameter ranges and default values

Parameter	Range
query length q_l (% of space side)	$5, 10,$ **15**$, 20, 25$
k	$1, 3,$ **5**$, 7, 9$
$\lvert P\rvert/\lvert O\rvert$	$0.1, 0.2, 0.5,$ **1**$, 2, 5, 10$
buffer size (% of the tree size)	$1, 2, 4, 8,$ **10** $16, 32, 64$
number of trajectory segments $\lvert q\rvert$	$2, 4, 6, 8, 10$
δ (% of space side)	$5, 10, 15, 20, 25$
R_c (% of space side)	$10, 20, 30, 40, 50, 60, 70$

cities and villages in Greece, respectively; *SLA* and *RGR* include 2D rectangles, representing $131,461$ MBRs of streets in Los Angeles and $24,650$ MBRs of rivers in Greece, respectively. All the datasets are normalized in order to fit the search range. Synthetic datasets are generated based on uniform distribution and zipf distribution respectively, with the cardinality varying from $0.1 \times |SLA|$ to $10 \times |SLA|$. The coordinate of each point in *Uniform* datasets is generated uniformly along each dimension, and that of each point in *Zipf* datasets is generated according to zipf distribution with skew coefficient $\alpha = 0.8$. We assume a point's coordinates on both dimensions are mutually independent.

Since CVNN query and its variants involve a data set P and an obstacle set O, we deploy four different combinations of the datasets, namely ***CR***, ***LS***, ***US***, and ***ZS***, representing (P, O) = (*CGR*, *RGR*), (*LCA*, *SLA*), (*Uniform*, *SLA*), and (*Zipf*, *SLA*), respectively. Both *CR* and *LS* utilize real datasets with $|P| < |O|$. On the other hand, *US* and *ZS* employ synthetic datasets, and we can adjust the relative size of P and O to simulate different cases. Consequently, we will explicitly specify the ratio of $|P|/|O|$ for *US* and *ZS* in the following evaluations. Note that the data points in P are allowed to lie on the boundaries of the obstacles, but not in their interior.

All data and obstacle sets are indexed by R*-trees [20], with a page size of 4K bytes. We employ an LRU memory buffer whose default size is set to 10% of the tree size. Table 4 lists all the parameters that are considered in the experiments, with numbers in bold representing default settings. In each experiment, only one parameter is changed in order to evaluate its impact on the performance, while all the other parameters are fixed at their defaults. We run 200 queries for each experiment, and the average performance is reported.

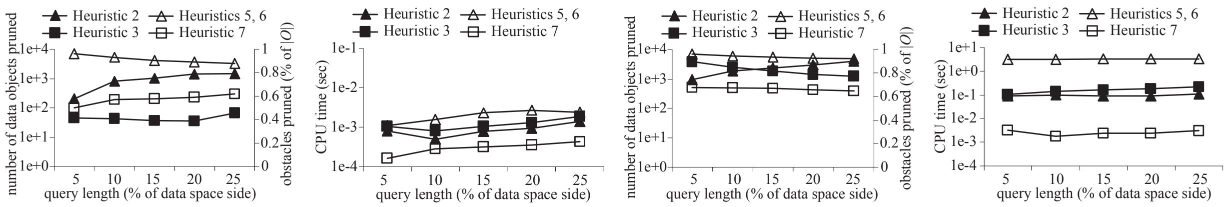
We consider I/O cost, CPU time, and total query cost as the main performance metric. Here, I/O cost refers to the number of pages/nodes accessed, and the query cost is the summation of the I/O time and CPU time where the I/O time is computed by charging 10ms for each page access [52]. Given a query length q_l, each query line segment is generated by (i) selecting a random point in the data space as the starting point of the query line segment, and (ii) selecting randomly an orientation (angle with the x-axis) from the range $[0, 2\pi)$, with its length controlled by the specified query length q_l. The line segments included into a predefined query trajectory for TVkNN search are generated in the same manner. However, we fix the trajectory length to 15% of the search space side, and assume all the line segments contained in the query trajectory share the same length in order to simplify the simulation.

7.2 Effectiveness of pruning heuristics

The first set of experiments aims at evaluating the effectiveness of the proposed pruning heuristics. As presented in Section 4, Heuristic 2 and Heuristic 3 prune away unnecessary node entries of the R-tree T_p on the dataset P, and thus we employ the number of data objects pruned as the performance metric. Similarly, Heuristic 5, Heuristic 6, and Heuristic 7 are developed to discard the obstacles that will certainly not affect the visibility of any potential answer object, and hence the percentage of obstacles pruned (account for the entire obstacle set O) is measured as the performance metric. Moreover, we measure the CPU time (in seconds) for the corresponding heuristic(s) to demonstrate their CPU cost. It is noticed that Heuristic 1 provides the early termination condition and Heuristic 4 specifies the *best-first* traversal fashion. Compared with other heuristics proposed, these two heuristics play a less significant role in pruning away data points or obstacles, and therefore their effectiveness is ignored from the evaluation. In addition, we illustrate the efficiency of Heuristic 5 and Heuristic 6 together because they are applied in the VRC algorithm simultaneously to prune unqualified obstacles (see line 3 of Algorithm 2).

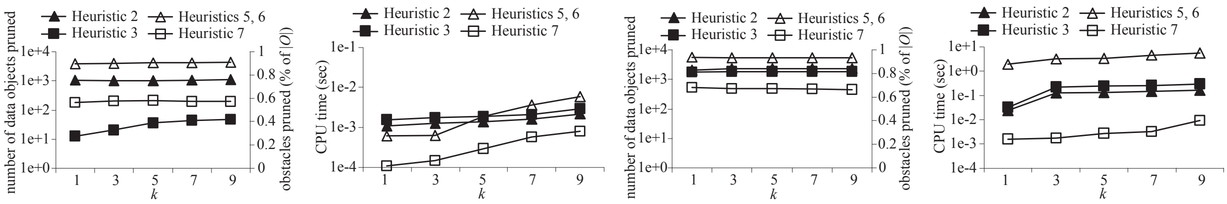
First, we vary the query length q_l of q from 5% to 25% of the side length of the search space, with k fixed at 5. The results are shown in Figures 22 for *CR* and *LS*, respectively. Evidently, each heuristic evaluated prunes away a large number of non-qualifying data objects or obstacles, which validates its usefulness. Take Heuristic 2 for *CR* combination as an example. It saves the detailed examination of 1,030 out of 5,922 objects when $q_l = 15\%$. Compared with Heuristic 3, Heuristic 2 has a more powerful pruning capability and a lower CPU overhead in most cases. This is because Heuristic 3 is more *strict*, i.e., it is *only* applied to those objects that cannot be pruned by Heuristic 2 via taking visibility into account. Similarly, for the pruning of the obstacles, Heuristics 5 and 6 are more effective than Heuristic 7, but incur a higher CPU cost since their implementation requires more computational overhead. Figure 23 and Figure 24 plot the pruning efficiency of different heuristics with respect to k and $|P|/|O|$ respectively, using dataset combinations *CR* and *LS*. The diagrams confirm the observations and corresponding explanations of Figure 22.

All the above experimental results are the average performance of *multiple* queries. In order to demonstrate the



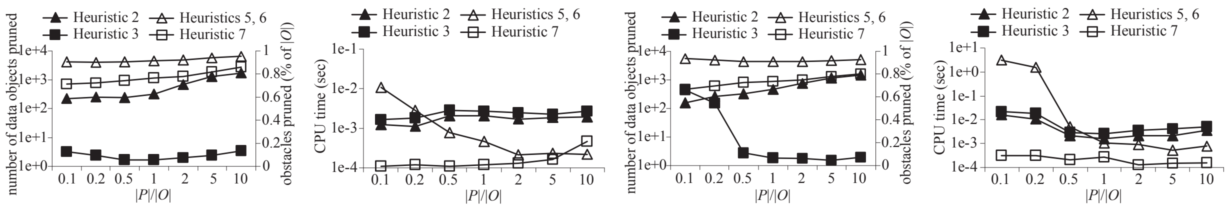
(a) CR (b) CR (c) LS (d) LS

Fig. 22 Pruning efficiency of heuristics versus q_l ($k = 5$)



(a) CR (b) CR (c) LS (d) LS

Fig. 23 Pruning efficiency of heuristics versus k ($q_l = 15\%$)



(a) US (b) US (c) ZS (d) ZS

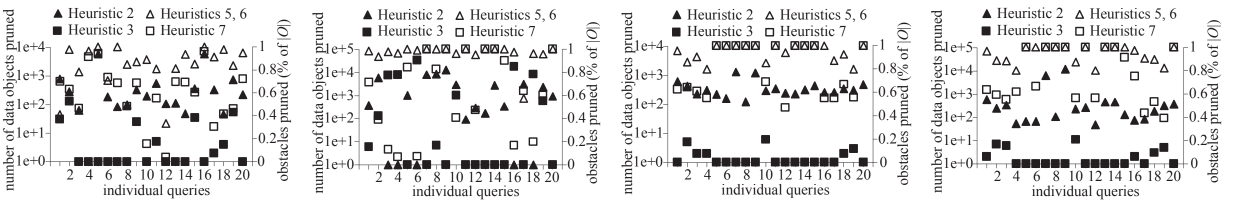
Fig. 24 Pruning efficiency of heuristics versus $|P|/|O|$ (q_l = 15%, k = 5)

pruning efficiency of heuristics on *individual* queries, Figure 25 shows the results on random 20 sample queries (i.e., 10% of each query workload) for different dataset combinations, by fixing q_l and k to their default values (i.e., 15% and 5, respectively). It is observed that although heuristics perform differently as queries change, their overall effectiveness is significant. In other words, all the evaluated heuristics are important because each heuristic is applied multiple times, and prunes unnecessary data objects or obstacles significantly, especially for Heuristics 2, 5, and 6.

7.3 Results on CVkNN queries

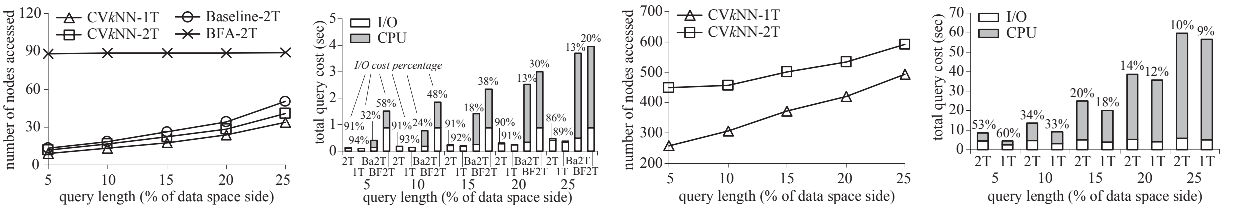
The second set of experiments evaluates the performance of CVNN algorithm in answering CVkNN queries. As CVNN search shares some similarities with CNN and VNN queries that have been well-studied, we implement two simple solutions extended from existing CNN and VNN search algorithms, i.e., Baseline and BFA, which are presented in Section 2.2 and Section 2.3 respectively. In our experiments, both k_1 and r of Baseline are set to $2k$ ($10k$). We study the influence of various parameters, including (i) query length q_l, (ii) the number of VNNs required k, (iii) the ratio of dataset cardinality $|P|$ to obstacle set cardinality $|O|$ (i.e., $|P|/|O|$), and (iv) buffer size. As explained in Section 5, P and O can be indexed either by two *separate* R-trees, denoted as 2T, or by a *single* R-tree, denoted as 1T. Note that Baseline and BFA can only support 2T case but not 1T scenario.

Effect of query length q_l. First, we investigate the impact of q_l on the efficiency of the algorithms, based on the dataset combinations *CR* and *LS* with k set to 5. The results are depicted in Figure 26. The first observation is that CVNN is several orders of magnitude faster than Baseline and BFA in all cases. The reason behind is that, as mentioned in Section 2, Baseline needs to invoke CkNN search *multiple* times and BFA requires scanning the *entire* dataset (in sequence) without any pruning. For presentation clarity, we skip Baseline and BFA in the remaining experimental results as CVNN consistently outperforms them. The second observation is that although CVkNN-2T and CVkNN-1T share a similar performance trend, CVkNN-1T performs better. This is because when data points and obstacles are indexed by one R-tree, only one traversal of the unified R-tree is required. Data points and obstacles that are close to each other could be found in the same leaf node of the R-tree. Hence, using a single R-tree to index P and O is one potential approach to further boost up the performance. It is also noticed that the cost of CVkNN queries increases with q_l. The reason behind is that, as the query length becomes longer, both the number of data points processed and the number of the split-



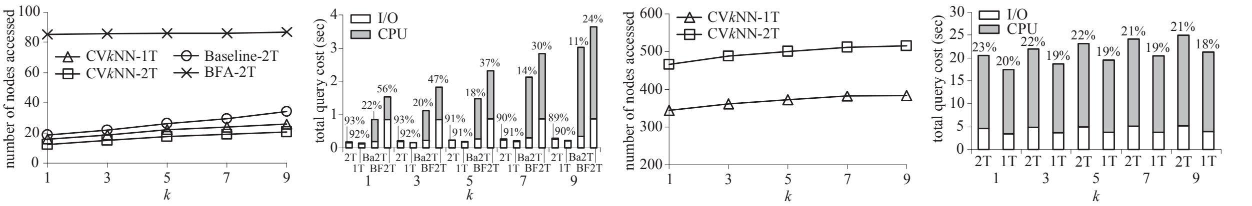
(a) CR (b) LS (c) US ($|P|/|O| = 1$) (d) ZS ($|P|/|O| = 1$)

Fig. 25 Pruning heuristic application efficiency of individual queries ($q_l = 15\%$, $k = 5$)



(a) CR (b) CR (c) LS (d) LS

Fig. 26 CVkNN search performance versus q_l ($k = 5$)



(a) CR (b) CR (c) LS (d) LS

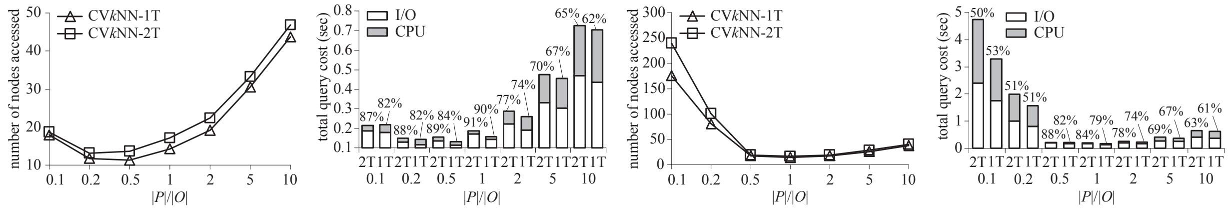
Fig. 27 CVkNN search performance versus k ($q_l = 15\%$)

ting regions in the specified query line segment increase, resulting in more distance computation, visibility examination, and result list updating. In addition, we find that the query cost on *LS* exceeds significantly that on *CR*. This is because *LS* contains far more data points than *CR* (i.e., in *CR*, $|P| = 5,922$, while in *LS*, $|P| = 62,556$.).

Effect of k. Next, we explore the impact of k and Figure 27 illustrates the performance of the CVNN algorithm under different k values with q_l fixed at 15%. Similar to what observed from previous experiment, CVkNN-1T is better than CVkNN-2T, and they share the similar performance trend. On the other hand, the value of k affects the performance. Moreover, it has a more significant impact on *CR* than on *LS*. This is caused by the different nature of the datasets. In *CR*, $|P|/|O| \approx 0.25$ while that in *LS* is around 0.5. In other words, the visibility of an object from *CR* on average is affected by more obstacles, compared against the object from *LS*. Hence, the dominance region of an answer object from *CR* is smaller, compared with that of an answer object from *LS*. As k grows, the efficiency of the CVNN algorithm for *CR* suffers from a more significant deterioration.

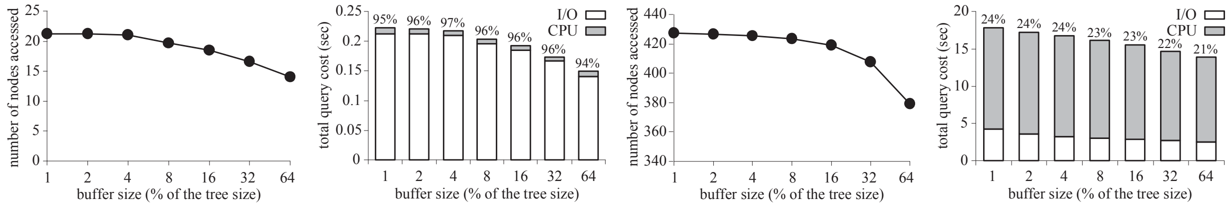
Effect of $|P|/|O|$. Then, we evaluate the performance of CVNN algorithm under different $|P|/|O|$ settings, with result plotted in Figure 28. As expected, CVkNN-1T outperforms CVkNN-2T and they share the similar performance trend. Consequently, we only present the performance of CVkNN-2T and ignore the performance of CVkNN-1T in the subsequent experiments. We also observe that as $|P|/|O|$ increases from 0.1 to 10, the cost of CVkNN queries first drops and then increases. This is because initially, the density of dataset P increases with the growth of $|P|/|O|$, which implies a smaller search range for the answer points and obstacles. Therefore, the performance improves. However, as $|P|/|O|$ further grows, the interval dominated by each data point becomes shorter, and the result list contains more answer points. The gain from the reduced search range cannot pay off the cost of frequent result list update operation, and thus the performance deteriorates. Notice that the performance of CVNN achieves the best performance when P and O share similar cardinalities (e.g., $|P|/|O| = 0.5$ or 1 in Figure 28).

Effect of buffer size. As mentioned in Section 7.1, all the above experiments are conducted with an LRU buffer that is set to 10% of the tree size. The fourth experiment examines the performance of CVNN with various LRU buffer sizes, by fixing $q_l = 15\%$ and $k = 5$. We use the first 100 queries to *warm up* the buffer, and the average cost of the last 100 queries is reported in Figure 29. Obviously, as the buffer size increases, the I/O cost drops gradually, whereas the CPU cost remains almost the same.



(a) US (b) US (c) ZS (d) ZS

Fig. 28 CVkNN search performance versus $|P|/|O|$ (q_l = 15%, k = 5)

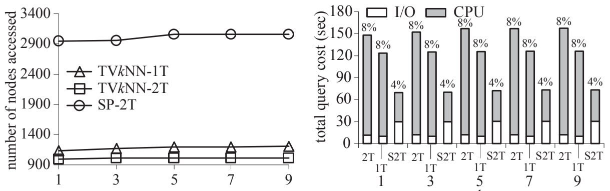


(a) CR (b) CR (c) LS (d) LS

Fig. 29 CVkNN search performance versus buffer size ($q_l = 15\%$, $k = 5$)

7.4 Results on TVkNN queries.

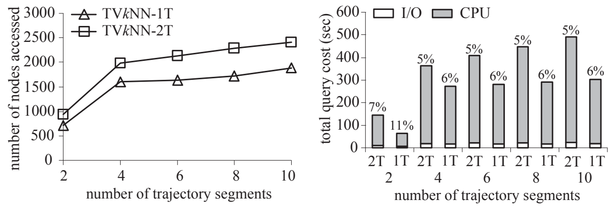
The third set of experiments evaluates the performance of CVNN algorithm in answering TVkNN queries. Here, the SP algorithm presented in Section 6.2 is taken as a baseline approach. We investigate the influence of two factors: k and the number of trajectory segments $|q|$. The trajectory length is set to 15% of the search space side length, and it consists of several connective line segments with equivalent length. Again, we consider the case where P and O are indexed by two separate R-trees and the case where P and O are indexed by one unified R-tree, denoted as TVkNN-2T (2T for short), TVkNN-1T (1T for short), and SP-2T (S2T for short), respectively.



(a) LS (b) LS

Fig. 30 TVkNN search performance versus k ($|q| = 3$)

First, we fix $|q|$ to 3 and vary k between 1 and 9 to study the effect of k on the efficiency of the algorithms, using the *LS* dataset combination. The experimental results are depicted in Figure 30. It is observed that the I/O cost of TVkNN outperforms significantly that of SP, but with a longer CPU time. The reason behind is that, as mentioned in Section 6.2, SP needs to traverse the data set and the obstacle set multiple times, resulting in numerous redundant node accesses; while TVkNN requires spending higher CPU time to implement pruning heuristics to avoid unnecessary node accesses. The second observation is that TVkNN-2T and TVkNN-1T share the similar performance trend, whereas TVkNN-1T performs better. As mentioned earlier, the advantage of TVkNN-1T can be explained by the fact that data points and obstacles located close to each other are very likely stored in the same page. Therefore, the access to the data points and that to the obstacles may share the node traversals when P and O are indexed by a single R-tree. In addition, the cost of TVkNN search increases as k grows, because a higher k value incurs a larger search space, more distance computation, and more result list maintenance cost.



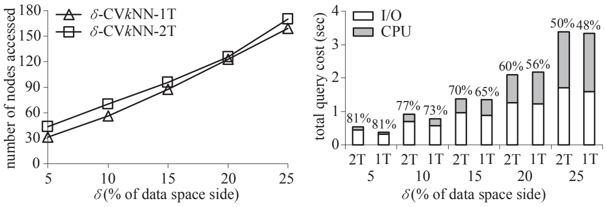
(a) LS (b) LS

Fig. 31 TVkNN search performance versus $|q|$ ($k = 5$)

Then, we explore the impact of $|q|$ on the performance of CVNN algorithm, with the result shown in Figure 31. As expected, TVkNN-1T outperforms TVkNN-2T and the cost of the algorithm increases with the growth of $|q|$.

7.5 Results on δ-CVkNN queries

The fourth set of experiments evaluates the performance of CVNN algorithm in answering δ-CVkNN queries. We vary the δ value from 5% to 25% of the search space side length, with q_l set to 15% and k fixed at 5. The experimental results are shown in Figure 32. Here, δ-CVkNN-2T (2T for short) represents the case where dataset P and obstacle set O are indexed by two different R-trees, and δ-CVkNN-1T (1T for short) denotes the case where P and O are indexed by a single R-tree.



(a) LS (b) LS

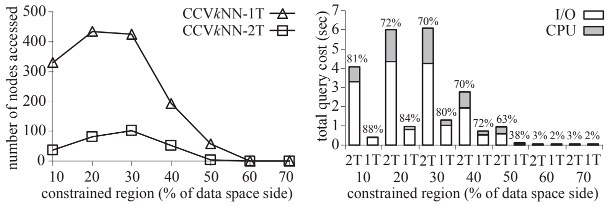
Fig. 32 δ-CVkNN search performance versus δ ($q_l = 15\%$, $k = 5$)

It is observed that δ-CVkNN-1T outperforms δ-CVkNN-2T, although their performance trend is similar. However, compared with the performance difference demonstrated in previous sets of experiments, the performance difference between δ-CVkNN-1T and δ-CVkNN-2T narrows down. The reason behind is that the search spaces for both data points and obstacles are bounded by δ, and thus the saving benefited from traversing one unified R-tree is less significant. On the other hand, as δ value grows, the cost of δ-CVkNN retrieval increases, which is consistent with our expectation and confirms that δ has a direct impact on the performance.

7.6 Results on CCVkNN queries

The last set of experiments evaluates the performance of CVNN algorithm in answering CCVkNN queries. We fix q_l and k to their default values (i.e., 15% and 5 respectively), and vary the constrained region R_c from 10% to 70% of the search space side length (i.e., from 1% to 49% of the search space area). Figure 33 plots the experimental results for the dataset combination *LS*, in which CCVkNN-2T (2T for short) represents the case where dataset P and obstacle set O are indexed by two separate R-trees, and CCVkNN-1T (1T for short) denotes the case where P and O are indexed by one unified R-tree.

Again, CCVkNN-1T demonstrates a better performance, while its performance trend is similar to that of CCVkNN-2T. It is observed that initially, the cost of CCVkNN search increases slightly with R_c, but thereafter it drops gradually as R_c further grows. The reason behind is that, when R_c is small (e.g., 10%, 20%), it is very likely that the answer objects to a traditional CVNN query are not located inside R_c, and hence nearly every data point within R_c has to be evaluated. If the query line segment q is far away from the constrained region R_c, more obstacles may affect the visibility of a data point inside R_c over q, resulting in a higher obstacle retrieval cost and a higher visible region formation cost. Consequently, as R_c increases, more data points need to be evaluated with a higher search cost and a higher I/O overhead. However, as R_c reaches a certain size (e.g., 60%, 70%), it is very likely that data points located close to q are inside R_c, and thus the search space that has to be traversed for retrieving the answer objects is reduced, which leads to a significant improvement of the search performance.



(a) LS (b) LS

Fig. 33 CCVkNN search performance versus R_c ($q_l = 15\%$, $k = 5$)

8 Conclusions

This paper identifies and solves a new type of spatial queries, namely *continuous visible nearest neighbor* (CVNN) search. CVNN retrieval is not only interesting from a research point of view, but also useful in many practical applications (involving spatial data and obstacles) such as decision support, mixed-reality games, and location-based commerce. We carry out a systematic study of CVNN queries. First, we provide a formal definition of the problem and reveal its unique characteristics. Second, we present a suite of effective pruning heuristics and develop an efficient CVNN algorithm to tackle the problem. Next, we extend CVNN algorithm to handle several CVNN query variants, including CVkNN, TVkNN, δ-CVkNN, and CCVkNN queries. Finally, we conduct extensive experiments to evaluate the effectiveness of the proposed pruning heuristics and the performance of the proposed algorithms.

In the future, we intend to explore the application of the proposed methodology to other forms of spatial queries (e.g., all nearest neighbor search, etc.) in the presence of obstacles. Another interesting direction for future work is to investigate visibility queries for moving objects and moving obstacles. Finally, it would be particularly interesting to develop analytical models for estimating the query cost of CVNN search and its variants, because such models will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even better algorithms.

References

1. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. In: SSTD, pp. 79–96 (2001)

2. Tao, Y., Papadias, D.: Time parameterized queries in spatio-temporal databases. In: SIGMOD, pp. 334–345 (2002)
3. Tao, Y., Papadias, D., Shen, Q.: Continuous nearest neighbor search. In: VLDB, pp. 287–298 (2002)
4. Nutanong, S., Tanin, E., Zhang, R.: Visible nearest neighbor queries. In: DASFAA, pp. 876–883 (2007)
5. Nutanong, S., Tanin, E., Zhang, R.: Incremental evaluation of visible nearest neighbor queries. IEEE Transactions on Knowledge and Data Engineering (to appear)
6. Gao, Y., Zheng, B., Chen, G., Lee, W.C., Lee, K.C.K., Li, Q.: Visible reverse k-nearest neighbor queries. In: ICDE, pp. 1203–1206 (2009)
7. 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. IEEE Transactions on Knowledge and Data Engineering **21**(9), 1314–1327 (2009)
8. Xia, C., Hsu, D., Tung, A.K.H.: A fast filter for obstructed nearest neighbor queries. In: BNCOD, pp. 203–215 (2004)
9. Zhang, J., Papadias, D., Mouratidis, K., Zhu, M.: Spatial queries in the presence of obstacles. In: EDBT, pp. 366–384 (2004)
10. Gao, Y., Zheng, B.: Continuous obstructed nearest neighbor queries in spatial databases. In: SIGMOD, pp. 577–590 (2009)
11. Estivill-Castro, V., Lee, I.: Autoclust+: Automatic clustering of point-data sets in the presence of obstacles. In: TSDM, pp. 133–146 (2000)
12. Park, S.H., Lee, J.H., Kim, D.H.: Spatial clustering based on moving distance in the presence of obstacles. In: DASFAA, pp. 1024–1027 (2007)
13. Tung, A.K.H., Hou, J., Han, J.: Spatial clustering in the presence of obstacles. In: ICDE, pp. 359–367 (2001)
14. Wang, X., Hamilto, H.J.: Clustering spatial data in the presence of obstacles. International Journal on Artificial Intelligence Tools **14**(1-2), 177–198 (2005)
15. Wang, X., Rostoker, C., Hamilton, H.J.: Density-based spatial clustering in the presence of obstacles and facilitators. In: PKDD, pp. 446–458 (2004)
16. Zaiane, O.R., Lee, C.H.: Clustering spatial data in the presence of obstacles: A density-based approach. In: IDEAS, pp. 214–223 (2002)
17. Sharifzadeh, M., Kolahdouzan, M., Shahabi, C.: The optimal sequenced route query. The VLDB Journal **17**(4), 765–787 (2008)
18. Sharifzadeh, M., Shahabi, C.: Processing optimal sequenced route queries using voronoi diagrams. GeoInformatica **12**(4), 411–433 (2008)
19. Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.H.: On trip planning queries in spatial databases. In: SSTD, pp. 273–290 (2005)
20. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: SIGMOD, pp. 322–331 (1990)
21. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD, pp. 47–54 (1984)
22. Gao, Y., Zheng, B., Lee, W.C., Chen, G.: Continuous visible nearest neighbor queries. In: EDBT, pp. 144–155 (2009)
23. Cheung, K.L., Fu, A.W.C.: Enhanced nearest neighbour search on the R-tree. SIGMOD Record **27**(3), 16–21 (1998)
24. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD, pp. 71–79 (1995)
25. Papadopoulos, A., Manolopoulos, Y.: Performance of nearest neighbor queries in R-trees. In: ICDT, pp. 394–408 (1997)
26. Henrich, A.: A distance-scan algorithm for spatial access structures. In: GIS, pp. 136–143 (1994)
27. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Transactions on Database Systems **24**(2), 265–318 (1999)
28. Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A.: Constrained nearest neighbor queries. In: SSTD, pp. 257–278 (2001)
29. Papadias, D., Shen, Q., Tao, Y., Mouratidis, K.: Group nearest neighbor queries. In: ICDE, pp. 301–312 (2004)
30. Papadias, D., Tao, Y., Mouratidis, K., Hui, K.: Aggregate nearest neighbor queries in spatial databases. ACM Transactions on Database Systems **30**(2), 529–576 (2005)
31. Zhang, J., Mamoulis, N., Papadias, D., Tao, Y.: All-nearest-neighbors queries in spatial databases. In: SSDBM, pp. 297–306 (2004)
32. Deng, K., Zhou, X., Shen, H., Xu, K., Lin, X.: Surface k-NN query processing. In: ICDE, p. 78 (2006)
33. Hu, H., Lee, D.L.: Range nearest-neighbor query. IEEE Transactions on Knowledge and Data Engineering **18**(1), 78–91 (2006)
34. Sistla, A.P., Wolfson, O., Chamberlain, S., Dao, S.: Modeling and querying moving objects. In: ICDE, pp. 422–432 (1997)
35. Iwerks, G.S., Samet, H., Smith, K.: Continuous k-nearest neighbor queries for continuously moving points with updates. In: VLDB, pp. 512–523 (2003)
36. Li, Y., Yang, J., Han, J.: Continuous k-nearest neighbor search for moving objects. In: SSDBM, pp. 123–126 (2004)
37. Mouratidis, K., Hadjieleftheriou, M., Papadias, D.: Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In: SIGMOD, pp. 634–645 (2005)
38. Xiong, X., Mokbel, M.F., Aref, W.G.: SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE, pp. 643–654 (2005)
39. Yu, X., Pu, K., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE, pp. 631–642 (2005)
40. Liu, F., Hua, K.A., Do, T.T.: A P2P technique for continuous k-nearest-neighbor query in road networks. In: DEXA, pp. 264–276 (2007)
41. Mouratidis, K., Yiu, M., Papadias, D., Mamoulis, N.: Continuous nearest neighbor monitoring in road networks. In: VLDB, pp. 43–54 (2006)
42. Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.: A threshold-based algorithm for continuous monitoring of k nearest neighbors. IEEE Transactions on Knowledge and Data Engineering **17**(11), 1451–1464 (2005)
43. Wu, W., Guo, W., Tan, K.L.: Distributed processing of moving k-nearest-neighbor query on moving objects. In: ICDE, pp. 1116–1125 (2007)
44. Zheng, B., Lee, W.C., Lee, D.L.: On searching continuous k nearest neighbors in wireless data broadcast systems. IEEE Transactions on Mobile Computing **6**(7), 748–761 (2007)
45. Feng, J., Watanabe, T.: A fast method for continuous nearest target objects query on road network. In: VSMM, pp. 182–191 (2002)
46. Kolahdouzan, M.R., Shahabi, C.: Alternative solutions for continuous k nearest neighbor queries in spatial network databases. GeoInformatica **9**(4), 321–341 (2005)
47. Cho, H.J., Chung, C.W.: An efficient and scalable approach to CNN queries in a road network. In: VLDB, pp. 865–876 (2005)
48. Asano, T., Ghosh, S.K., Shermer, T.C.: Visibility in the plane. In: Handbook of Computation Geometry. Elsevier (2000)
49. Kofler, M., Gervautz, M., Gruber, M.: R-trees for organizing and visualizing 3D GIS databases. Journal of Visualization and Computer Animation **11**(3), 129–143 (2000)
50. Shou, L., Chionh, C., Ruan, Y., Huang, Z., Tan, K.L.: Walking through a very large virtual environment in real-time. In: VLDB, pp. 401–410 (2001)
51. Shou, L., Huang, Z., Tan, K.L.: HDoV-tree: The structure, the storage, the speed. In: ICDE, pp. 557–568 (2003)
52. Tao, Y., Papadias, D., Lian, X., Xiao, X.: Multidimensional reverse kNN search. The VLDB Journal **16**(3), 293–316 (2007)