5-2011

Continuous Nearest Neighbor Search in the Presence of Obstacles

Yunjun GAO

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

Gang CHEN

Chun CHEN

Qing LI

Continuous Nearest Neighbor Search in the Presence of Obstacles

X

YUNJUN GAO
Zhejiang University
BAIHUA ZHENG
Singapore Management University
GANG CHEN and CHUN CHEN
Zhejiang University
and
QING LI
City University of Hong Kong

Despite the ubiquity of physical obstacles (e.g., buildings, hills, and blindages, etc.) in the real world, most of spatial queries ignore the obstacles. In this article, we study a novel form of continuous nearest neighbor queries in the presence of obstacles, namely *continuous obstructed nearest neighbor* (CONN) search, which considers the impact of obstacles on the *distance* between objects. Given a data set P, an obstacle set O, and a query line segment q in a two-dimensional space, a CONN query retrieves the nearest neighbor $p \in P$ of each point p' on q according to the *obstructed distance*, i.e., the *shortest path* between p and p' without crossing any obstacle in O. We formalize CONN search, analyze its unique properties, and develop algorithms for *exact* CONN query processing assuming that both P and O are indexed by conventional data-partitioning indices (e.g., R-trees). Our methods tackle CONN retrieval by performing a *single* query for the *entire* query line segment, and only process the data points and obstacles *relevant to the final query result* via a novel concept of *control points* and an efficient *quadratic-based split point computation approach*. Then, we extend our techniques to handle variations of CONN queries, including (i) *continuous obstructed k nearest neighbor* (CO*k*NN) search which, based on obstructed distances, finds the k (≥ 1) nearest neighbors (NNs) to every point along q; and (ii) *trajectory obstructed k nearest neighbor* (TO*k*NN) search which, according to obstructed distances, returns the k NNs for each point on an arbitrary trajectory (consisting of several consecutive line segments). Finally, we explore *approximate* CO*k*NN (ACO*k*NN) retrieval. Extensive experiments with both real and synthetic datasets demonstrate the efficiency and effectiveness of our proposed algorithms under various experimental settings.

Categories and Subject Descriptors: H.2.8 **[Database Management]**: Database Applications—*Spatial databases and GIS*; H.2.4 **[Database Management]**: Systems—*Query processing*

General Terms: Algorithms, Design, Experimentation, Performance

---

Y. Gao was supported in part by NSFC Grant 61003049, ZJNSF Grant Y1100278, the Fundamental Research Funds for the Central Universities under Grant No. 2010QNA5051, the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan), and NBNSF Grant 2010A610113.
Authors' addresses: Y. Gao, College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China; email: gaoyj@zju.edu.cn; B. Zheng, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore; email: bhzheng@smu.edu.sg; G. Chen (corresponding author), College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China; email: cg@zju.edu.cn; C. Chen, College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China; email: chenc@zju.edu.cn; Q. Li, Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong; email: itqli@cityu.edu.hk.

DOI XX.XXXX/XXXXXXX.XXXXXXX http://doi.acm.org/XX.XXXX/XXXXXXX.XXXXXXX

Additional Key Words and Phrases: Nearest neighbor, continuous nearest neighbor, obstacle, spatial databases

ACM Reference Format:

Gao, Y., Zheng, B., Chen, G., Chen, C., and Li, Q. 20XX. Continuous nearest neighbor search in the presence of obstacles. ACM Trans. Datab. Syst. XX, X, Article XX (XXXXXXXX 20XX), XX pages.
DOI = XX.XXXX/XXXXXXX.XXXXXXX http://doi.acm.org/XX.XXXX/XXXXXXX.XXXXXXX

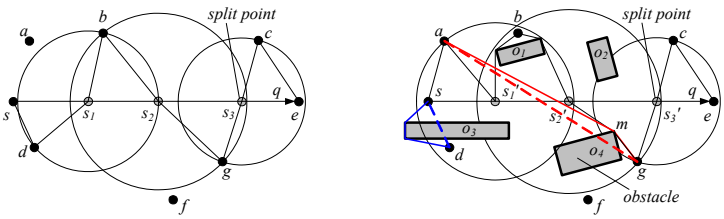
1. INTRODUCTION

The mobile communication revolution is in full swing. It is reported that the world reached over four billionth mobile connections in early 2009, and the number of global wireless connections is expected to reach six billion in 2013. More and more customers are experiencing the incredible benefits and conveniences only offered by wireless communication, and they can access information anywhere even on the move. Consequently, *continuous* queries are proposed to cater for the dynamic nature of the queries issued by clients who are moving [Terry et al. 1992; Chen et al. 2000]. Compared with traditional *snapshot* queries of constant settings, continuous queries require continuous evaluation and real-time updates of the results as the query settings change.

Continuous nearest neighbor (CNN) search is such an example. It retrieves from a given data set P the nearest neighbor (NN) of every point on a specified query line segment q. Example applications for CNN queries include "report the nearest restaurant" submitted by a hungry tourist shopping on the Fifth avenue, and "find the closest exit" issued by a lost driver driving along the Highway 401. The result of a CNN query contains a set of $\langle p, I \rangle$ tuples, such that $p \in P$ is the NN of all points along the interval $I \subseteq q$. Figure 1(a) illustrates an example CNN query, in which the data set $P = \{a, b, c, d, f, g\}$ and the query line segment $q = [s, e]$. The output of the CNN query is $\{\langle d, [s, s_1] \rangle, \langle b, [s_1, s_2] \rangle, \langle g, [s_2, s_3] \rangle, \langle c, [s_3, e] \rangle\}$, meaning that point d is the NN for any point along the interval $[s, s_1]$, point b is the NN for any point along the interval $[s_1, s_2]$, and so on. The points s_1, s_2, s_3 on q where there is a change of the NN object are called *split points*.

Existing work on CNN search utilizes either *Euclidean distance* or *network distance* to measure the proximity of objects [Song and Roussopoulos 2001; Tao and Papadias 2002; Tao et al. 2002; Feng and Watanabe 2002; Kolahdouzan and Shahabi 2005; Cho and Chung 2005]. However, there are applications where neither Euclidean distance nor network distance can represent the closeness between objects. For instance, battlefields usually have no fixed road network structure and soldiers would enjoy certain degrees of free movement until they hit obstacles (e.g., blindages, etc.). Another example is that mobile robots help rescue survivors after a disaster (e.g., a devastating earthquake). The robots equipped with location-sensing ability as well as visual and other sensors can burrow into the rubble and try to locate potential survivors, which can facilitate the excavation without further injuring survivors. Theoretically, the robot navigating the space can take any direction, but the existence of physical obstacles (e.g., rocks, etc.) affects the real distance that a robot has to travel in order to reach its destination. To this end, in this article, we study a novel form of CNN queries in an *obstructed space*, namely, *continuous obstructed nearest neighbor* (CONN) search, where the object movement is constrained by the obstacles.

Compared with the Euclidean space, an obstructed space considers the presence of obstacles that may block the immediate path from one object to another and hence the Euclidean distance between them does not always indicate the actual travelling distance. On the other hand, compared with the network space, it does not assume any underlying



(a) CNN search (b) CONN search

Fig. 1. Example of CNN and CONN queries

fixed network structure and still entitles the objects to free movement. Correspondingly, the distance between two objects in the obstructed space is measured based on the *obstructed distance*, i.e., the length of the shortest path connecting two objects without crossing any obstacle. Take the points a and g shown in Figure 1(b) as an example. Their Euclidean distance is the length of the line segment $[a, g]$, whereas their obstructed distance is the summation of the lengths of the line segment $[a, m]$ and the line segment $[m, g]$, because of the obstruction of obstacle o_4.

Given a data set P, an obstacle set O, and a query line segment[1] q in a 2-dimensional space, a CONN query retrieves the NN of each point on q according to the obstructed distance, i.e., the *obstructed nearest neighbor* (ONN) [Zhang et al. 2004; Xia et al. 2004] for every point along q. Specifically, CONN retrieval aims to find a set of $\langle p, R\rangle$ tuples, where $p \in P$ is the ONN for any point in the interval $R \subseteq q$. Continue the example in Figure 1(b), with $P = \{a, b, c, d, f, g\}$, $O = \{o_1, o_2, o_3, o_4\}$ (denoted by shaded rectangles[2]), and $q = [s, e]$. Note that the placement of data points and query line segment q in Figure 1(b) is the same as that in Figure 1(a). The CONN query returns $\{\langle a, [s, s_1']\rangle, \langle b, [s_1', s_2']\rangle, \langle g, [s_2', s_3']\rangle, \langle c, [s_3', e]\rangle\}$, which means that point a is the ONN for each point along interval $[s, s_1']$, point b is the ONN for each point along interval $[s_1', s_2']$, and so forth. Also notice that, the split points (i.e., s_1, s_2, and s_3) of the CNN search are different from those (i.e., s_1', s_2', and s_3') generated by the CONN retrieval. Moreover, their *answer points*[3] vary as well. For instance, point d is the NN to point s in a Euclidean space, whereas it is not the ONN of s in an obstructed space, due to the block of obstacle o_3.

CONN search is useful for many applications. Consider an example application scenario. A zoologist in wildlife reserve may want to find the nearest observation points where he/she can most likely observe a certain rare animal (e.g., panda) along his/her trail[4], defined by a starting point s and an ending point e. However, there may exist some obstacles (e.g., hills, rivers, etc.) on his/her route so that he/she has to detour these obstacles to reach the destination. In this case, a CONN query can be employed to find out the nearest observation point(s) for each (sub) route along a specified zoologist traveling route. Although conventional CNN retrieval can also be applied to this scenario, the result of CONN search provides more accurate information in terms of distance, since the CONN query considers obstacles. In addition, in view of the ubiquity of physical obstacles in the real world, the CONN query is obviously important, as a stand-alone tool or a stepping stone, in location-based commerce, geographic information systems, mobile

[1]In this article, we assume that any query line segment does not intersect/cross any obstacle.
[2]Although an obstacle might be in any shape (e.g., triangle, etc.), we assume it is a rectangle in this article.
[3]In the rest of this article, we refer to the data objects/points in the final query result as *answer objects*/*points*.
[4]While a trail may not be a straight line, it can be decomposed into multiple line segments.

computing, and complex spatial data analysis/mining involving obstacles.

To answer efficiently CONN search, two challenging issues have to be addressed.

Challenge I: How to calculate the obstructed distance efficiently? Based on the existing work related to robot motion planning, the lower bound of obstructed distance computation is $O(n \cdot \log n)$, where n is the total number of obstacle vertexes [Berg et al. 2000]. Although an asymptotically optimal algorithm has been devised in [Hershberger and Suri 1999], it is very complex and has a large hidden constant, which makes it impractical. In practice, a popular and practical method based on a *visibility graph VG* [Berg et al. 2000] has $O(n^2 \cdot \log n)$ as the worst case time complexity. Compared with the Euclidean distance computation that can be completed in constant time, the calculation cost of the obstructed distance is much more expensive. Therefore, it is not feasible to compute the obstructed distance from the query point to every data point. Furthermore, *VG*-based approaches need to maintain a visibility graph, which requires $O(n^2)$ space in the worst case. The high space complexity deteriorates its scalability, not to mention its extremely high update cost. Thus, it is not always applicable to store the *whole VG* in main memory or *pre-materialize VG* (as in [Tung et al. 2001a]).

We try to tackle this issue from two aspects, i.e., reducing the number of obstructed distance calculations and simplifying the computation of obstructed distances. The first objective is achieved via effective *pruning techniques* that can filter out unqualified data points and obstacles as early as possible. In other words, we prune away a large amount of non-qualifying data points and obstacles and only process/evaluate the data points and obstacles *relevant to the final query result*. As for the second target, we construct a *local VG* to simplify the process of obstructed distance calculation. Initially, the local visibility graph only contains two endpoints of a given query line segment. As we process the query and evaluate data points, we incrementally insert the obstacles that may affect the obstructed distances from the query point to the evaluated points into the local visibility graph. Owing to *the small size of the local VG*[5] (to be demonstrated by our experimental results), the insertion/deletion/update operation is efficient.

Challenge II: How to answer a CONN query efficiently? A naive approach is to perform an ONN query [Zhang et al. 2004] at every point of a specified query line segment q. However, this method is *infeasible* since the number of points on q is *infinite*. Given the fact that nearby points along the query line segment may share the same ONN, we adopt an incremental approach to fine-tune the result upon the evaluation of each new data point using the concept of split point. Nevertheless, due to the existence of obstacles, existing split point formation algorithms developed for CNN search [Tao et al. 2002] cannot be applied directly. Hence, we introduce, in this article, a novel concept, namely *control point*, to facilitate the obstructed distance computation, and develop a *quadratic-based* approach to identify split points. Specifically, we utilize a quadratic function to compute the split points, and exploit quadratic characteristics to quickly determine the intervals (bounded by the split points) for which a data point is closer to than another. Moreover, several pruning strategies and optimizations are proposed to further improve the search performance.

In addition to CONN retrieval, in this article, we also investigate two variations of CONN queries, including (i) *continuous obstructed k nearest neighbor* (CO*k*NN) search, which retrieves the k (≥ 1) obstructed nearest neighbors (ONNs) of every point on a given

[5]The size of the local visibility graph *VG* depends on several factors, including the distribution/position of a specified query line segment q, the length of q, and the distribution of a given data set P and a given obstacle set O. Note that, in the worst case, the size of the local *VG* might be similar to that of the complete *VG*.

query line segment; and (ii) *trajectory obstructed k nearest neighbor* (TO*k*NN) search, where the query input is an *arbitrary trajectory*[6] (instead of a *single line segment*), and the goal is to find the *k* ONNs for every point along the query trajectory. In addition, as opposed to the *exact* CO*k*NN query, *approximate* CO*k*NN (ACO*k*NN) retrieval is explored in this article as well. Correspondingly, our techniques are extended to support CO*k*NN search, TO*k*NN search, and ACO*k*NN search, respectively.

In brief, the key contributions of this article can be summarized as follows.

— We formalize CONN search, a new addition to the family of spatial queries with obstacle constraints. To our knowledge, this work is the first attempt on this problem.

— We introduce the concept of control point that significantly simplifies the computation of the obstructed distance between two objects, and develop a quadratic-based method to identify split points by solving quadratic inequalities.

— We propose an efficient CONN search algorithm, which handles the CONN retrieval by performing a *single* query for the *whole* query line segment and only processes the data points and obstacles *relevant to the final query result*.

— We tackle variants of CONN queries, including CO*k*NN search and TO*k*NN search, and present an efficient approach to answer ACO*k*NN retrieval.

— We conduct extensive experimental evaluation using both real and synthetic datasets to demonstrate the efficiency and effectiveness of the proposed algorithms under a variety of experimental settings.

The rest of this article is organized as follows. Section 2 surveys related work. Section 3 formulates the CONN query, introduces the concept of control point, and presents the split point computation approach. Section 4 proposes efficient CONN query processing algorithms, assuming that the data set *P* and the obstacle set *O* are indexed by two *separate* R-trees [Guttman 1984; Sellis et al. 1987; Beckmann et al. 1990] and a *unified* R-tree, respectively. Section 5 extends our solution to handle CONN query variants (viz., CO*k*NN search and TO*k*NN search). ACO*k*NN retrieval is discussed in Section 6. Section 7 experimentally evaluates the performance of the proposed algorithms, and reports experimental results and our findings. Finally, Section 8 concludes the article with some directions for future work.

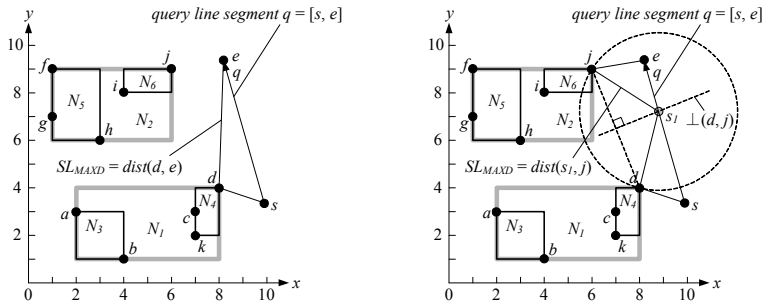
2. RELATED WORK

In this section, we review the existing work related to CONN queries, namely, CNN search, spatial queries under obstacle constraints, and main-memory based obstacle paths.

2.1 Continuous Nearest Neighbor Search

With the proliferation of mobile e-commerce and mobile computing, CNN search becomes increasingly important. Sistla et al. [1997], for the first time, identify the significance of CNN retrieval in spatio-temporal databases. In that pioneering work, they describe modeling methods and query languages for the expression of CNN queries, but do not present processing methods. Song and Roussopoulos [2001] propose the first algorithm for CNN query processing by employing periodical sampling technique. In particular, the method incrementally computes the output at predefined sample points of the query line segment, and utilizes previous results to avoid total recomputation. However, its performance highly depends on the sampling rate, and thus the accuracy

[6]In this article, for simplicity, we model a trajectory with a sequence of line segments. If the trajectory is a *smooth curve*, we can model it with a number of short line segments.



(a) After processing *d* (b) After processing *j*

Fig. 2. Example of CNN algorithm

cannot be guaranteed. Specifically, a low sampling rate improves the performance but may result in incorrect results, whereas a high sampling rate incurs high computational overhead but decreases the possibility of producing incorrect results. In any case, there is no accuracy guarantee since even a high sampling rate may also miss some results. Therefore, the sampling based approach cannot be adopted to tackle *exact* CONN search, the focus of this article.

In order to enable exact CNN retrieval, two algorithms for CNN queries, using R-tree as the underlying index structure, are proposed [Tao and Papadias 2002; Tao et al. 2002]. The first algorithm [Tao and Papadias 2002] is based on the concept of *time-parameterized* (TP) queries, which treats a query line segment as the moving trajectory of a query point. Hence, the closest 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 current closest object expires, i.e., when a split point is reached. Although the approach avoids the drawbacks of sampling, it needs to issue m TP queries to obtain the final query result, where m is the number of split points along the query line segment. Such a recomputation leads to prohibitive computational cost. In order to improve the search performance, the second algorithm, proposed later in [Tao et al. 2002], retrieves all the answer objects for the *whole* query line segment by navigating R-tree only once, which is also the objective of our solution (i.e., the proposed CONN algorithm) presented in this article.

Since the CONN algorithm proposed in this article and CNN algorithm presented in [Tao et al. 2002] share the similar principle, we illustrate the basic idea of CNN algorithm using a running example. Note that they aim at different queries and have different split point computation approaches. As shown in Figure 2, a CNN query is issued at a line segment $q = [s, e]$, with a dataset $P = \{a, b, c, d, f, g, h, i, j, k\}$. While the CNN algorithm developed in [Tao et al. 2002] supports both the *best-first* [Henrich 1994; Hjaltason and Samet 1999] and *depth-first* [Roussopoulos et al. 1995; Cheung and Fu 1998] traversal paradigms, we assume that, for simplicity, the data points in P are visited in a best-first fashion, i.e., those closer to q are evaluated earlier. For each evaluated point $p \in P$, the algorithm finds the set of points along q that are covered by p, i.e., being closest to p, discards the points that will not cover any point on q, and fine-tunes the *covering relationship*[7] during the traversal.

[7]If a point p is the nearest neighbor (NN) of any point along the interval $[s_i, s_{i+1}]$, we say that p *covers* every point on $[s_i, s_{i+1}]$. Please refer to [Tao et al. 2002] for the details of covering relationship.

Initially, the result list is set to $\{\langle \varnothing, [s, e]\rangle\}$, which indicates that the entire query line segment q is not covered by any data point, and the pruning metric SL_{MAXD} that maintains the maximal distance between any point along q and its current NN is set to ∞. Then, the traversal of P starts. When point d, the first data point visited, is evaluated, it covers the whole query line segment. Thus, the result list is updated to $\{\langle d, [s, e]\rangle\}$, and SL_{MAXD} is changed to $dist(d, e)$[8], as illustrated in Figure 2(a). Next, point j is evaluated. As it is closer to e than its current NN (i.e., d), the result list is updated to $\{\langle d, [s, s_1]\rangle, \langle j, [s_1, e]\rangle\}$ and SL_{MAXD} is set to $dist(s_1, j)$, as depicted in Figure 2(b). Note that, s_1 is the intersection between the query line segment q and the perpendicular bisector of line segment $[d, j]$ (denoted as $\perp (d, j)$), meaning that points along segment $[s, s_1]$ are closer to d, while points along segment $[s_1, e]$ are closer to j (see Figure 2(b)). Thereafter, point c is evaluated. Since its minimal distance to q exceeds the current SL_{MAXD}, c will not invalidate the current covering relationship of any answer point and hence can be pruned away safely. Here, the algorithm terminates because all the unexamined entries are guaranteed to have their minimum distances to q greater than SL_{MAXD}.

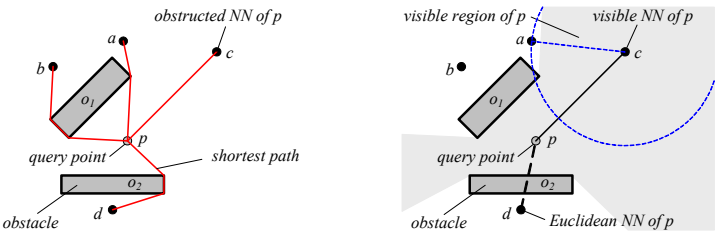
In addition to Euclidean spaces, CNN search in road networks (i.e., network spaces) has been studied as well. The first solution to CNN queries in road networks is proposed in [Feng and Watanabe 2002]. Nevertheless, it only answers CNN retrieval and does not support CkNN ($k > 1$) search. In view of this, Kolahdouzan and Shahabi [2005] present two methods to address CkNN queries in road networks, namely, *Intersection Examination* (IE) and *Upper Bound Algorithm* (UBA). IE first finds the k NNs of all nodes on a path and then, for those adjacent nodes whose NNs are different, it finds the intermediate split points, and finally, it computes the k NNs of the split points using the k NNs of the surrounding nodes. The intuition behind UBA is that if an object moves slightly, its k NNs will probably remain the same. Consequently, UBA improves the performance of IE via restricting the evaluation of kNN queries to those necessary locations only. The same problem is also solved in [Cho and Chung 2005], by retrieving the kNN sets of all network nodes in the query path and combining them with the data objects falling in the path. It can be easily proven that the resulting set contains the k NNs of any point in the query path. Note that, the problem of CNN search in road networks is inherently different from CONN retrieval studied in this article. First, it focuses on the network space, where objects are restricted to move only on pre-defined trajectories/paths that are specified by the underlying network (e.g., road, railway, etc.). This means that the network distance between objects depends on the connectivity of the network rather than the object locations. Second, it does not consider the impact of obstacles in terms of distance. Consequently, the existing methods for CNN search in road networks are inapplicable to CONN retrieval.

Recently, the CNN *monitoring* problem, which monitors the answer objects to a CNN query for a given duration, has also been investigated. Based on the concept of *monitoring region*, many monitoring algorithms (e.g., CPM [Mouratidis et al. 2005a], SEA-CNN [Xiong et al. 2005], and YPK-CNN [Yu et al. 2005]) have been proposed. Here, the monitoring region corresponding to a query point refers to an area inside which the movement of objects might affect the query result. Thus, those objects that are always outside the monitoring region could be safely discarded. Other variants of the CNN monitoring include (i) CNN monitoring in road networks [Mouratidis et al. 2006; Liu et al. 2007], where the distance between two objects is defined as their shortest path length; (ii) CNN monitoring in distributed environments [Mouratidis et al. 2005b; Wu et al.

[8]Without loss of generality, $dist(p_i, p_j)$ denotes the Euclidean distance between points/objects p_i and p_j.

2007], where the optimization target is to reduce the communication cost between the central query processor and the data objects; and (iii) CNN monitoring over sliding windows [Mouratidis and Papadias 2007], where data arrive in the form of streams, and each data item is valid only while it belongs to a sliding window. In addition, C*k*NN retrieval over moving objects has been explored in [Iwerks et al. 2003; Li et al. 2004; Huang et al. 2009] as well. It is worthwhile to point out that, these works assume that data objects are moving while the query object is static/moving. Our problem studied in this article (i.e., CONN search), on the other hand, assumes that the query object is moving while data objects are static. Moreover, these works target at the Euclidean space but do not take into account the obstacle space, where the obstacle distance (instead of the Euclidean distance) is employed to indicate the proximity between objects. Hence, the existing approaches for them are not directly applicable to CONN retrieval.

Another closely related work is M*k*NN query that finds the *k* NNs of a *moving query point*. A common practice is based on *safe region*; that is an area associated with an answer set within which the answer set remains valid for current M*k*NN query. In other words, *safe-region-based* approaches for M*k*NN query locate the answer set based on the current location of moving query point, and derive corresponding safe region. As long as the query point stays inside the safe region, the answer remains valid. Once the query point moves out of the safe region, another answer with its associated safe region is returned. For example, *Voronoi Diagram* [Okabe et al. 2000] is an example of safe region, and can be used to tackle the M1NN query, which involves (i) locating which Voronoi cell the specified query point falls into; and (ii) identifying the associated object. A natural generalization of Voronoi Diagram is the *kth-order Voronoi Diagram* (*k*VD), which can be utilized to handle M*k*NN queries in the same manner. Unfortunately, they both have several deficiencies, including (i) expensive precomputation, (ii) inefficient update operations, and (iii) no support for dynamical changing of *k*. Subsequently, Zhang et al. [2003] develop an algorithm, namely *Retrieve-Influence-Set kNN* (RIS-*k*NN), to locally compute the *k*VD using a spatial index (e.g., R-tree). RIS-*k*NN mitigates the above precomputation and update problems (i.e., deficiencies (i) and (ii)). However, it still does not support dynamical changing of *k*. Recently, Nutanong et al. [2008] propose a novel approach, called *V*-Diagram*, which is a safe-region-based technique for processing M*k*NN queries. The *V*-Diagram* addresses all the above three deficiencies, i.e., it requires no precomputation, adapts to changes in the dataset, and supports dynamically changing *k* values. A good survey of the safe-region-based techniques for handling M*k*NN queries can be found in [Nutanong et al. 2010]. It is worth noting that, the above safe-region-based techniques (for M*k*NN queries) differ from our approach for CONN search proposed in this article. First, an obvious advantage of the safe-region-based techniques is that the query trajectory/path does not need to be known in advance, i.e., the location of query object is updated in a periodic manner. In contrast, our work presented in this article assumes that the query trajectory is known in advanced, represented by a line segment or a set of line segments. Second, the safe-region-based techniques continuously produce answers, whereas our approach returns all answers for a given query trajectory only once. Third, the safe-region-based techniques ignore the impact of obstacles in terms of distance, which has to be considered in our work. Thus, the existing safe-region-based techniques are not directly applicable to CONN retrieval. Note that, it would be an interesting and challenging topic to investigate the safe-region-based methods for answering CONN search. However, we would like to leave it to our future work due to the focus of this article (i.e., how to efficiently find out all the answer objects for the *whole* query line segment or query trajectory by navigating R-tree only



(a) ONN search (b) NN, VNN, and VRNN queries

Fig. 3. Example of NN, ONN, VNN, and VRNN queries

once, as pointed out earlier) and the space limitation.

2.2 Spatial Queries under Obstacle Constraints

The existence of obstacles could affect the distance, visibility, and spatial clustering of spatial objects. First, in terms of distance, Zhang et al. [2004] and Xia et al. [2004] study the *obstructed nearest neighbor* (ONN) query where, given a data set P, an obstacle set O, and a query point p, the goal is to find the k (≥ 1) objects in P that have the smallest obstructed distances from p. An example ONN query issued at point p is depicted in Figure 3(a), with $P = \{a, b, c, d\}$ and $O = \{o_1, o_2\}$. Point a having the minimal obstructed distance to p is the final answer point. Existing ONN search algorithm utilizes conventional NN retrieval to retrieve objects close to the query point as candidates, and terminates when the retrieved NN object has its Euclidean distance to the query point larger than the maximum obstructed distance of the candidates. Apart from ONN search, Zhang et al. [2004] also propose algorithms for processing other popular spatial queries, including range search, e-distance joins, and closest pairs, in the presence of obstacles. More recently, Li et al. [2010] propose the concept of obstacle-free safe region for handling the MkNN query in the obstructed space. They assume, however, *unknown* query trajectory instead of *known* one that is assumed in this article.

A CONN query processing algorithm has been developed in [Gao and Zheng 2009]. Nonetheless, this article extends the initial study, via (i) formulating and tackling two variations of CONN queries, namely, *continuous obstructed k nearest neighbor* (COkNN) search that retrieves the k (≥ 1) ONNs of every point on a given query line segment and *trajectory obstructed k nearest neighbor* (TOkNN) search which returns the k ONNs for every point along a specified query trajectory consisting of several consecutive line segments; (ii) formalizing *approximate* COkNN (ACOkNN) search and proposing a method for fast ACOkNN retrieval; (iii) conducting a more comprehensive performance evaluation that incorporates the new classes of queries; and (iv) providing a more complete review of the related work, and including more illustrative examples, and more formal proofs.

Second, in terms of visibility, two objects are visible to each other iff the straight line segment connecting them does not pass through any obstacle. Nutanong et al. [2007] introduce *visible nearest neighbor* (VNN) search to find the NN object that is visible to a given query point. An example of VNN retrieval issued at a query point p is illustrated in Figure 3(b), where the shaded area represents the visible region of p within which all the objects/points are visible to p. Point c that is the only point inside p's visible region is the answer point. A VNN query algorithm, based on the fact that a faraway object cannot affect the visibility of a nearby object, is proposed in [Nutanong et al. 2007]. The basic

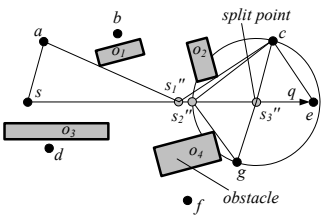


Fig. 4. Example of CVNN search

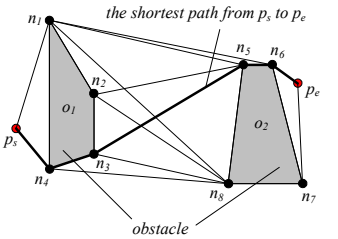


Fig. 5. Visibility graph and obstacle path

idea is to perform NN search and check visibility condition in an incremental manner. However, the algorithm can only support VNN retrieval for a fixed point but not a line segment. Later, Gao et al. [2009a] explore *visible reverse nearest neighbor* (VRNN) search where, given a data set P, an obstacle set O, and a query point p, the goal is to retrieve all the points in P that have p as their VNN. Take a VRNN query issued at point p as an example (shown in Figure 3(b)). Since no single point takes p as its VNN, the VRNN result set is *empty*. An algorithm for VRNN query processing, assuming that both P and O are indexed by R-trees, is presented in [Gao et al. 2009a]. The algorithm follows a filter-refinement framework, and requires no pre-processing. Specifically, it identifies a candidate set during the filter step, and filters out false hits in the refinement step, with these two steps integrated into a single R-tree traversal. Furthermore, pruning techniques based on half-plane properties (as [Tao et al. 2007]) and visibility check are developed to further improve the search performance. Along this line, Gao et al. [2009b] also study *continuous visible nearest neighbor* (CVNN) search where, given a data set P, an obstacle set O, and a query line segment q, the goal is to return the VNN of every point on q. Consider, for example, Figure 4, which uses the same data set $P = \{a, b, c, d, f, g\}$, obstacle set $O = \{o_1, o_2, o_3, o_4\}$, and query line segment $q = [s, e]$ as Figure 1(b). The CVNN query returns $\{\langle a, [s, s_1'']\rangle, \langle c, [s_1'', s_2'']\rangle, \langle g, [s_2'', s_3'']\rangle, \langle c, [s_3'', e]\rangle\}$, indicating that point a is the VNN for any point along interval $[s, s_1'']$, point c is the VNN for any point along interval $[s_1'', s_2'']$, and so on. In [Gao et al. 2009b], a CVNN search algorithm, assuming that both P and O are indexed by R-trees, is proposed. The basic idea is to traverse data points in P based on ascending order of their *mindist*[9] to q. For each data point $p \in P$ visited, the algorithm evaluates p's impact on the current query result. Like CONN retrieval, CVNN search takes obstacles into consideration and assumes a known query line segment. However, they are fundamentally different. First, they adopt different distance metrics. CONN retrieval employs obstructed distance to measure the distance between objects. On the other hand, CVNN search utilizes Euclidean distance to indicate the proximity of objects. Second, their query results are different. CONN retrieval returns a set of $\langle p, R\rangle$ tuples, where point p is the ONN for each point on the interval R; while the output of CVNN search is a set of $\langle p', R'\rangle$ tuples, in which point p' is the VNN for every point on the interval R'. It is important to note that, if a given data set P is not empty, p can not be empty but p' might be empty due to the obstruction of obstacles. Last but not the least, CONN retrieval focuses on the impact of obstacles in terms of distance, whereas CVNN search considers the impact of obstacles in terms of visibility. Therefore,

[9]The *mindist* is a distance metric. Specifically, *mindist*(p, N) corresponds to the minimal Euclidean distance between a data point/object p and any data point/object in (the subtree of) node N. Please refer to [Roussopoulos et al. 1995; Hjaltason and Samet 1999] for details.

CONN retrieval differs from CVNN search, and algorithms developed for CVNN retrieval could not be directly applied to answer CONN search. More recently, Xu et al. [2010] explore *group visible nearest neighbor* (GVNN) where, given a data set P, an obstacle set O, and a query set Q, the goal is to retrieve $p \in P$ such that (i) p is visible to any query point $q_i \in Q$, and (ii) $\Sigma_{qi \in Q}\, dist(p, q_i)$ is minimized.

Third, the problem of *spatial clustering in the presence of obstacles* has attracted considerable attention in recent years. It divides a set of 2D data points into smaller homogeneous groups (i.e., clusters) by considering the impact of obstacles. Handling these constraints can lead to effective and fruitful data mining by capturing application semantics [Tung et al. 2001b]. A large number of clustering algorithms with obstacle constraints have been developed in the literature, including AUTOCLUST+ [Estivill-Castro and Lee 2000], COD_CLARANS [Tung et al. 2001a], DBCLuC [Zaiane and Lee 2002], DBRS+ [Wang et al. 2004], DBRS_O [Wang and Hamilton 2005], and DBSCAN_MDO [Park et al. 2007], etc.

2.3 Main-Memory Based Obstacle Paths

Main-memory based shortest path problem in the presence of obstacles has been well-studied in computational geometry [Berg et al. 2000], and the most common approach is based on the visibility graph VG. A VG is constructed based on an obstacle set O and the source/destination point p_s/p_e. Its nodes correspond to the vertexes of the obstacles or source/destination point. Two nodes n_i, n_j are connected if and only if the straight line segment connecting them does not cross any obstacle interior.

An example VG is shown in Figure 5, where shaded polygons represent obstacles. Nodes n_2 and n_7 are not connected directly as the corresponding straight line segment [n_2, n_7] intersects with obstacle o_2. There are multiple paths available from the source point p_s to the destination point p_e, such as the path via nodes n_1, n_6 and the path via nodes n_1, n_8, and n_7. Among all the available paths, the one with the shortest distance is returned, i.e., the path via nodes n_4, n_3, n_5 and n_6 (denoted by thick line in Figure 5). Since the shortest path contains only the edges of VG (as proved in [Berg et al. 2000]), a popular and practical obstacle path (i.e., shortest path) computation method proceeds in two steps. The first step constructs VG, which takes $O(n^2 \cdot \log n)$ based on *rotational plane sweep* [Sharir and Schorr 1986], and can be optimized to $O(m + n \cdot \log n)$ with an optimal output-sensitive algorithm [Ghosh and Mount 1987]. Here, n is the number of nodes in VG and m is the number of edges in VG. The second step computes the shortest path in VG using Dijkstra's algorithm [Dijkstra 1959], which incurs $O(m + n \cdot \log n)$ time. Thus, the time and space complexities of the approach are $O(n^2 \cdot \log n)$ and $O(n^2)$, respectively. Obviously, the method has a poor scalability and cannot guarantee the efficiency when a large number of obstacles are considered. As mentioned in Section 1, we, in this article, efficiently calculate the obstacle distance (i.e., the shortest path length) by reducing the number of obstructed distance calculations and simplifying the computation of obstructed distances. In addition, the calculation of the shortest path distance with obstacles has also been discussed in [Okabe et al. 2000], using the *shortest path voronoi diagram with obstacles*.

3. PRELIMINARIES

In this section, we provide the formal definition of CONN search, introduce the concept of control point, and present the quadratic-based split point computation approach that is crucial to our proposed CONN search algorithm. Table I summarizes the notations used in the rest of this article.

Table I. Symbols and descriptions

Notation	Description
P	The set of data points p in a two-dimensional space
O	The set of obstacles o in a two-dimensional space
T_p	The R-tree on P
T_o	The R-tree on O
q	The query line segment with $q = [s, e]$
VG	The visibility graph
RL	The result list of a CONN query

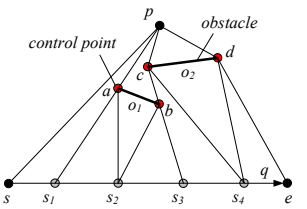


Fig. 6. Example of control point list

3.1 Problem Formulation

Given a set of data points $P = \{p_1, p_2, \ldots, p_n\}$, a set of obstacles $O = \{o_1, o_2, \ldots, o_m\}$, and a query line segment $q = [s, e]$ in a two-dimensional space, the problem of CONN retrieval is formulated as follows.

Definition 3.1 (*Visibility* [Gao et al. 2009a]). Given $p, p' \in P$ and O, 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 as $[p, p']$, crosses o, i.e., $\forall\ o \in O$, $[p, p'] \cap o = \varnothing$.

Definition 3.2 (*Visible Region*). Given $p \in P$ and q, the *visible region* of p over q, denoted by $VR_{p,q}$, is defined as the set of intervals $R \subseteq q$ such that p is *visible* to all the points along R.

In a Euclidean space, any two objects are visible to each other because there are no obstacles. However, this statement does not necessarily hold in an obstructed space. As shown in Figure 6, the visible region $VR_{p,q}$ of p over q is interval $[s, s_1]$, and the rest (i.e., interval $[s_1, e]$) is blocked by obstacles[10] o_1 and/or o_2. Since point s_2 on q is not located inside p's visible region over q, i.e., $s_2 \notin VR_{p,q}$, it is *invisible* to point p. The visible region computation algorithm has been proposed in [Gao et al. 2009a; Gao et al. 2009b].

Definition 3.3 (*Obstacle-Free Path*). Given $p, p' \in P$ and O, a path connecting p and p' sequentially passes n nodes $d_1, d_2, \ldots, d_n$ (i.e., n vertexes of obstacles in O), denoted as $P(p, p') = \{d_1, d_2, \ldots, d_n\}$. Let $d_0 = p$, $d_{n+1} = p'$, and assume $P(p, p')$ reaches d_i before d_{i+1}. $P(p, p')$ is an *obstacle-free path* (path for short) iff $\forall\ i \in [0, n]$, d_i and d_{i+1} are *visible* to each other. Its length $|P(p, p')| = \sum_{i \in [0, n]} dist(d_i, d_{i+1})$.

Definition 3.4 (*Shortest Obstacle-Free Path*). Given $p, p' \in P$, the *shortest obstructed-free path* (shortest path for short) from p to p', denoted by $SP(p, p')$, is the (obstacle-free) path connecting p and p' that has the *smallest* length, i.e., $\forall\ P(p, p')$, $|P(p, p')| \geq |SP(p, p')|$.

Definition 3.5 (*Obstructed Distance* [Tung et al. 2001a]). Given $p, p' \in P$, the *obstructed distance* between p and p', denoted by $||p, p'||$, is defined as the length of the shortest (obstacle-free) path $SP(p, p')$ from p to p', i.e., $||p, p'|| = |SP(p, p')|$.

Given a set of obstacles, there are usually multiple obstacle-free paths from a specified point p to another point p'. As an example, in Figure 6, the path $P(p, e) = \{c, b\}$ passes c and b before reaching e; and $P(p, e) = \{d\}$ provides an alternative obstacle-free path connecting p and e. Among all the obstacle-free paths from p to e, the one with the

[10]For simplicity, we utilize line segments but not rectangles to represent obstacles in the rest of this article, although the proposed techniques are applicable to rectangles that consist of multiple line segments.

minimal distance, i.e., $P(p, e) = \{d\}$, is the shortest path $SP(p, e)$. The obstructed distance between p and e is the length of the shortest path that connects p and e, i.e., $||p, e|| = |SP(p, e)| = dist(p, d) + dist(d, e)$.

Definition 3.6 (*Obstructed Nearest Neighbor*). Given $p' \in P$ and $p \notin P$, p' is the *obstructed nearest neighbor* (ONN) of p iff $\forall\, p'' \in P$, $||p', p|| \leq ||p'', p||$.

Definition 3.7 (*Continuous Obstructed Nearest Neighbor Query*). Given P, O, and q, a *continuous obstructed nearest neighbor* (CONN) *query* returns the result list RL that contains a set of $\langle p_i, R_i\rangle$ ($i \in [1, t]$) tuples, such that (i) $\cup_{i\in[1,t]} R_i = q$; (ii) $\forall\, i, j \in [1, t]$ ($i \neq j$ and $|i - j| \neq 1$, i.e., R_i and R_j are *discontinuous*), $R_i \cap R_j = \emptyset$; (iii) $\forall\, i \in [1, t)$, R_i (= $[R_i.l, R_i.r]$) $\cap$ R_{i+1} (= $[R_{i+1}.l, R_{i+1}.r]$) = $R_i.r$; and (iv) $\forall \langle p_i, R_i\rangle \in RL$, p_i is the ONN of every point along interval R_i.

In this article, we focus on efficient processing of CONN search and its variations.

3.2 Split Point Computation

As mentioned in Section 1, a naive CONN query processing approach is to perform ONN retrieval at every point of a given query line segment q. Unfortunately, it is not feasible due to the *unlimited* number of points on q. It is observed that nearby points along q normally share the same ONN. Take a result list RL (= $\cup_{i\in[1,t]} \langle p_i, R_i\rangle$) for a CONN query as an example. The object p_i is the ONN for each point on interval R_i. Thus, it is only necessary to issue ONN search at those points where ONN objects change. In view of this, the concept of split point is introduced, as defined in Definition 3.8.

Definition 3.8 (*Split Point for CONN Search*). Given two points $p, p' \in P$ ($p \neq p'$) and $q = [s, e]$, if p is the ONN to any point along $[s, s_1]$ and p' is the ONN for every point on $[s_1, e]$, point s_1 is called a *split point* where the ONN corresponding to q changes.

Based on the concept of split point, the CONN retrieval can be conducted as follows. Initially, none of the data points is evaluated and the result list $RL = \langle \emptyset, q\rangle$. When the first data point p is evaluated, p is definitely the ONN for any point along q, i.e., $RL = \{\langle p, q\rangle\}$. As more and more data points are processed, split points are generated and q will be decomposed into several smaller segments/intervals with each having its own ONN. In other words, the evaluation of a new data point p' is converted to check whether the presence of p' introduces any new split point on a region/interval R_i contained in the current result list RL. Nevertheless, due to the existence of obstacles, the computation of split points for the CONN query is not *trivial*, and it is different from that for CNN search. To facilitate the formation of split points, in this article, we introduce a novel concept, namely, *control point*, which is defined in Definition 3.9.

Definition 3.9 (*Control Point*). Given $p \in P$, O, and an interval R, a point cp is the *control point* of p over R, denoted as $CP_{p,R}$, iff (i) the shortest path from p to any point on R passes through cp; and (ii) cp is *visible* to every point along R.

Note that, the control point cp of a given data point p over a specified interval R is either the data point p or a vertex of the obstacle that locates on the shortest path from p to any point on R. As illustrated in Figure 6, all the points on interval $R = [s_1, s_2]$ have their shortest paths to p passing through point a, and a is visible to every point along R. Thus, point a is the control point of point p over the interval R, i.e., $CP_{p,R}$, according to Definition 3.9. Based on the concept of control point, each point p has its *control point list* over q, denoted as $CPL_{p,q}$, formalized in Definition 3.10. In order to utilize the

concept of control point to facilitate the computation of obstructed distance, we reformat the result list RL into $\cup_{i\in[1,\ t]} \langle p_i, cp_i, R_i\rangle$ such that point p_i is the ONN of every point along interval R_i, and point cp_i is the corresponding control point of p_i over R_i. The detection algorithm for control points will be presented in Section 4.

Definition 3.10 (*Control Point List*). Given $p \in P$ and q, the *control point list* of p over q, denoted by $CPL_{p,q}$, contains a set of $\langle cp_i, R_i\rangle$ ($i \in [1, t']$) tuples, such that (i) $\cup_{i\in[1,\ t']} R_i = q$; (ii) $\forall\ i, j \in [1, t']$ ($i \neq j$ and $|i - j| \neq 1$), $R_i \cap R_j = \varnothing$; (iii) $\forall\ i \in [1, t')$, R_i (= $[R_i.l$, $R_i.r]$) $\cap$ R_{i+1} (= $[R_{i+1}.l$, $R_{i+1}.r]$) = $R_i.r$; and (iv) $\forall\ \langle cp_i, R_i\rangle \in CPL_{p,q}$, cp_i is the *control point* of p over interval R_i.

In the following, we will explain the rationale behind the concept of control points, i.e., to facilitate the identification of split points and to provide pruning opportunity. Given two points p, p' and a line segment q, suppose point v is the control point of p over q, point u is the control point of p' over q, and $||p, v||$, $||p', u||$ are known with $||p, v|| - ||p', u|| = d$. We further assume that p is the ONN for every point along q before p' is accessed, and now we need to evaluate p'. In other words, we need to check whether p' is closer to any point on q, compared with p. If yes, we have to locate the split points along q where ONN is changed from p to p'. Although the locations of u, v and the value of d have a direct impact on the number and positions of the split points that are introduced by p', it is confirmed that the *maximum number* of split points generated by p' is *two*, as stated in Theorem 3.1. Moreover, based on Theorem 3.1, we can easily determine the positions of split points on q by solving quadratic inequalities.

THEOREM 3.1. *Given two points p, p', a line segment q = $[s, e]$, together with corresponding control point v (u) of p (p') over q, let $d = ||p, v|| - ||p', u||$. There are* at most two *points s_1, s_2 on q having the same obstructed distances to p and p', i.e., $||p, v|| + dist(v, s_1) = ||p', u|| + dist(u, s_1)$ and $||p, v|| + dist(v, s_2) = ||p', u|| + dist(u, s_2)$.*

PROOF. Please refer to Appendix A. □

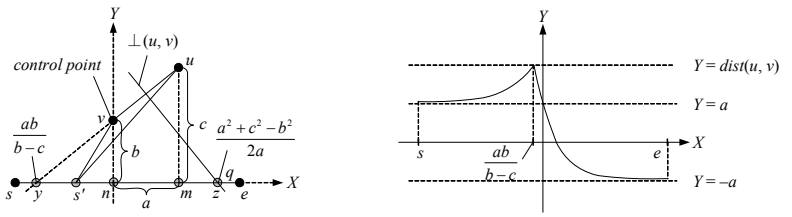
Theorem 3.1 demonstrates that there are at most two distinct points along q such that their obstructed distances to points p and p' are equivalent. We can also prove that when q is decomposed into several smaller intervals R by split point(s) sp on q, all the points along R must share the same ONN (either p or p'). In order to facilitate understanding, we transfer the Equation (1) (which is presented in Appendix A) to the following Equation (2), and assume point s' is located at (x, 0). The positions of split points correspond to the x values such that $Y(x) = d$. Figure 7(b) illustrates the distribution of $Y(x)$ under different values of x.

$$Y(x) = dist(u,s') - dist(v,s') = \sqrt{(a-x)^2 + c^2} - \sqrt{x^2 + b^2} \tag{2}$$

Based on the derivative and the limit of Equation (2) with respect to the variable x, as shown in Equation (3), we find that (i) when $x = ab/(b-c)$, $Y'(x) = 0$ and $Y(x)$ reaches its maximal value[11] $dist(u, v)$, which can also be verified by triangle inequality[12]; (ii) when $x < ab/(b-c)$, $Y'(x) > 0$, which indicates that $Y(x)$ is *monotone increasing*, and $Y(x) \in (a, dist(u, v))$; and (iii) when $x > ab/(b-c)$, $Y'(x) < 0$, which means that $Y(x)$ is *monotone*

[11]Note that the distribution of $Y(x)$ under other cases (e.g., $a = 0$, $b > c$, etc.) has different trend, i.e., different inflexion point(s) and maximal/minimal value.

[12]Based on Figure 7(a) and triangle inequality, it is clear that $dist(u, s') - dist(v, s') < dist(u, v)$ as long as points u, v and s' form a triangle (i.e., they are not located along a line). Consequently, when u, v and s' are located along one line, $Y(x) = dist(u, s') - dist(v, s')$ reaches its maximal value $dist(u, v)$, i.e., $x = ab/(b-c)$.



(a) Example (b) $Y(x) = dist(u, s') - dist(v, s')$

Fig. 7. Properties of split points

decreasing, and $Y(x) \in (-a, dist(u, v))$. The positions of split point(s) on q can be determined as follows, according to the value of $d = ||p, v|| - ||p', u||$ and $Y(x)$.

$$Y'(x) = \frac{x-a}{\sqrt{(a-x)^2 + c^2}} - \frac{x}{\sqrt{x^2 + b^2}}$$
$$\lim_{x \to +\infty} Y(x) = -a \text{ , and } \lim_{x \to -\infty} Y(x) = a \tag{3}$$

Case 1: $d \geq dist(u, v)$. As $Y(x) \leq dist(u, v)$, it is certain that, for any point s' along q, $Y(x) = dist(u, s') - dist(v, s') \leq dist(u, v) \leq d = ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') \leq ||p, v|| + dist(v, s')$. Thus, new point p' will replace point p (that is the current ONN for every point on q) as the ONN for any point along q without introducing any new split point.

Case 2: $a < d < dist(u, v)$. As depicted in Figure 7(b), there will be *two* values x_1 and x_2 such that $Y(x_1) = Y(x_2) = d$, with $x_1 < ab/(b-c) < x_2$. Let $(x_1, 0)$ be s_1 and $(x_2, 0)$ be s_2. For a given point s' with coordinate $(x, 0)$, (i) if $x < x_1$ or $x > x_2$, $Y(x) < d$, meaning that $dist(u, s') - dist(v, s') < ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') < ||p, v|| + dist(v, s')$, and hence point p' becomes the ONN for each point along intervals $[s, s_1]$ and $[s_2, e]$; and (ii) if $x_1 \leq x \leq x_2$, $Y(x) \geq d$, indicating that $dist(u, s') - dist(v, s') \geq ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') \geq ||p, v|| + dist(v, s')$, and thus point p is still the ONN for every point on interval $[s_1, s_2]$. In this case, p' introduces *two* split points s_1 and s_2.

Case 3: $-a < d \leq a$. As shown in Figure 7(b), there will be *only one* value x_1 such that $Y(x_1) = d$. Let $(x_1, 0)$ be s_1. For a specified point s' with coordinate $(x, 0)$, (i) if $x < x_1$, $Y(x) > d$, which means that $dist(u, s') - dist(v, s') > ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') > ||p, v|| + dist(v, s')$, and hence point p is still the ONN for any point along interval $[s, s_1]$; and (ii) if $x \geq x_1$, $Y(x) \leq d$, which indicates that $dist(u, s') - dist(v, s') \leq ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') \leq ||p, v|| + dist(v, s')$, and thus point p' becomes the ONN for every point on interval $[s_1, e]$. In this case, p' introduces *only one* split point s_1.

Case 4: $d \leq -a$. As $Y(x) > -a$, $dist(u, s') - dist(v, s') > d = ||p, v|| - ||p', u||$, i.e., $||p', u|| + dist(u, s') \geq ||p, v|| + dist(v, s')$. Therefore, point p is still the ONN of each point along q without introducing any new split point.

In the above discussion, we define a quadratic polynomial whose root(s) can be used to derive the number and position(s) of split point(s). However, some special case of **Case 1**/**Case 4** can be detected by Lemma 3.1 below, without expensive calculation of the quadratic polynomial. Its pruning power will be discussed in Section 4, where we present the CONN search algorithm.

LEMMA 3.1. *Given two points p, p', a line segment q = [s, e], together with corresponding control point v (u) of p (p') over q, suppose* $dist_{\perp}(u, q) > dist_{\perp}(v, q)$, *in which* $dist_{\perp}(u, q)$ ($dist_{\perp}(v, q)$) *denotes the vertical distance*[13] *from a control point u (v) to q. Point p is certainly closer to any point along q compared to p', if it satisfies* (i) $||p', u|| + dist(u, s) > ||p, v|| + dist(v, s)$; *and* (ii) $||p', u|| + dist(u, e) > ||p, v|| + dist(v, e)$.

PROOF. Pease refer to Appendix B. □

It is worth pointing out that, if $dist_{\perp}(u, q) < dist_{\perp}(v, q)$, we can also develop a lemma that is similar to Lemma 3.1. Based on Lemma 3.1, we introduce a *pruning distance*, namely, RL_{MAX} (see Lemma 3.2). As demonstrated by Lemma 3.2, if all the unexamined objects have their minimal Euclidean distances (i.e., *mindist*) to the query line segment *q* larger than RL_{MAX}, it is guaranteed that the current result list *RL* will not be updated by any unexamined object. In other words, Lemma 3.2 offers an *early termination condition*, which will be employed by our proposed CONN search algorithm in this article.

LEMMA 3.2. *Suppose the current result list RL for a CONN query issued at q = [s, e] is* $\cup_{i\in[1, t]} \langle p_i, cp_i, R_i\rangle$, *with interval* $R_i = [R_i.l, R_i.r] \subseteq q$. *Given a data point p, p cannot change RL if* $mindist(p, q) > RL_{MAX}$, *where* $RL_{MAX} = MAX_{i\in[1, t]}(||p_i, R_i.l||, ||p_i, R_i.r||)$[14].

PROOF. Pease refer to Appendix C. □

4. CONN QUERY PROCESSING

In this section, we propose efficient algorithms for CONN query processing, assuming that the data set *P* and the obstacle set *O* are indexed by two *separate* R-trees. The basic idea is to traverse data points in *P* in ascending order of their *mindist* (that is the lower bound of obstructed distances) to a given query line segment *q* = [*s*, *e*]. For each data point $p \in P$ visited, we first find all the obstacles that may affect the obstructed distances from *p* to any point along *q*, then identify the control point(s) of *p* over *q*, and finally evaluate the impact of *p* on the current result list *RL* which is initialized to $\langle \varnothing, \varnothing, [s, e]\rangle$. In what follows, we first elaborate the above three steps in Sections 4.1, 4.2, and 4.3, respectively. Then, we present the complete CONN search algorithm in Section 4.4, together with the analysis of its characteristics and correctness. Finally, we discuss how to adjust our techniques to tackle the CONN retrieval when *P* and *O* are indexed by one *unified* R-tree in Section 4.5.

4.1 Obstacle Retrieval

As pointed out in Section 1, the existing *VG*-based approach of obstructed distance calculation needs to maintain the visibility graph *VG*, and its high space and time complexities deteriorate its practicability. In general, for a specified data point *p* and a given query line segment *q* = [*s*, *e*], only a small number of obstacles will affect the obstructed distances between *p* and any point along *q*. It is worth noting that, the actual number of obstacles that affect the obstructed distance between *p* and any point along *q* depends on several factors, including the position of *p*, the distribution/position of *q*, and the length of *q*. As demonstrated in Theorem 4.1, once the shortest path from *p* to *s* (i.e., *SP*(*p*, *s*)) and that from *p* to *e* (i.e., *SP*(*p*, *e*)) are identified, the *search range* for all the obstacles that may affect the obstructed distances from *p* to any point along *q* is

[13]The vertical distance between a point *p* and a line segment *l*, denoted as $dist_{\perp}(p, l)$, refers to the Euclidean distance from *p* to the projection of *p* on the line segment *l* (or *l*'s extended line).

[14]If $\exists \langle p_i, cp_i, R_i\rangle \in RL$ with $p_i = \varnothing$, $||p_i, R_i.l|| = ||p_i, R_i.r|| = \infty$, and $MAX(p_i, p_j)$ is a function to return (i) p_i if $p_i \geq p_j$ or (ii) p_j otherwise.

Algorithm 1 Incremental Obstacle Retrieval Algorithm (IOR)

Input: an obstacle R-tree T_o; a min-heap H_o; a query line segment $q = [s, e]$;
a data point p; a visibility graph VG; previous search distance d

```
 1: while (1) do
 2:    P1(p, s) = Dijkstra(VG, p, s)    // compute the current short path P1(p, s)
 3:    P2(p, e) = Dijkstra(VG, p, e)    // compute the current short path P2(p, e)
 4:    d' = MAX(1/2 · (MIN(|P1(p, s)|, |P2(p, e)|) + |q| + mindist(p, q)), |P1(p, s)|, |P2(p, e)|)
 5:    if d' > d then
 6:       d = d'    // for the next loop
 7:       while Ho ≠ ∅ do
 8:          de-heap the top entry (e, key) of Ho
 9:          if key > d then    // key = mindist(e, q)
10:             break
11:          else if e is an obstacle then
12:             add e to a set So and insert all vertexes of e into VG
13:          else    // e is an intermediate (i.e., a non-leaf) node
14:             for each child entry ei ∈ e from To do
15:                insert (ei, mindist(ei, q)) into Ho
16:    else
17:       break
```

determined; and thus only those obstacles that intersect the identified search range have to be accessed.

THEOREM 4.1. *Given a data point p, a query line segment $q = [s, e]$, and the shortest path $SP(p, s)$ ($SP(p, e)$) from p to s (e), assume $d = 1/2 \cdot (\text{MIN}(||p, s||, ||p, e||) + |q| + mindist(p, q))$, where $||p, s||$ ($||p, e||$) denotes the obstructed distance from p to s (e), and $|q|$ represents the length of the query line segment q. Let $SR_{p,q}$ be the range within which all the objects have their minimal Euclidean distances (i.e., mindist) to q bounded by d. It is guaranteed that all the obstacles* outside *$SR_{p,q}$ will not affect the obstructed distance between p and any point along q.*

PROOF. Pease refer to Appendix D. □

In order to use Theorem 4.1 to bound the search range for all the obstacles affecting the obstructed distances from a specified data point p to any point along $q = [s, e]$, both $SP(p, s)$ and $SP(p, e)$ have to be identified. In view of this, Lemma 4.1 is proposed.

LEMMA 4.1. *Given a data point p, a point s' on q, and a path $P(p, s')$ from p to s', suppose all the obstacles o in O that have their minimal Euclidean distances (i.e., mindist) to q bounded by $|P(p, s')|$ have been retrieved and maintained in a set S_o, i.e., $S_o = \{o \in O \mid mindist(o, q) \le |P(p, s')|\}$. Let $P_2(p, s')$ be the shortest path from p to s' obtained based on S_o. If $|P_2(p, s')| \le |P(p, s')|$, it is confirmed that $P_2(p, s')$ must be the* real shortest path *$SP(p, s')$ between p and s', i.e., $P_2(p, s') = SP(p, s')$.*

PROOF. Pease refer to Appendix E. □

Based on Theorem 4.1 and Lemma 4.1, the *Incremental Obstacle Retrieval Algorithm* (IOR) is developed, with its pseudo-code depicted in Algorithm 1. The basic idea is to form the local visibility graph VG via incremental obstacle retrieval such that VG contains all the obstacles that might affect the obstructed distance from the point p evaluated currently to any point along the given query line segment q. As outlined in

Algorithm 1, based on the local *VG*, the local shortest paths from p to endpoints s and e, denoted as $P_1(p, s)$ and $P_2(p, e)$, can be identified respectively by Dijkstra's algorithm [Dijkstra 1959] (lines 2-3). Since the local *VG* contains these three points p, s, e, and all the vertexes of the obstacles which may affect $P_1(p, s)$ or/and $P_2(p, e)$ (via expanding incrementally the local *VG* until no triggering the retrieval of any new obstacle), we can easily find $P_1(p, s)$ and $P_2(p, e)$ based on the local *VG*, using Dijkstra Algorithm. Then, IOR fetches all the obstacles having their *mindist* to q bounded by $d' = \text{MAX}(1/2 \cdot (\text{MIN}(|P_1(p, s)|, |P_2(p, e)|) + |q| + mindist(p, q)), |P_1(p, s)|, |P_2(p, e)|)$, and inserts their vertexes into *VG* (lines 4-15). If *VG* is changed, both $P_1(p, s)$ and $P_2(p, e)$ are re-located, which may trigger the update of the radius d' and hence the retrieval of new obstacles. The process repeats until the new $P_1(p, s)$ and $P_2(p, e)$ have their distances to s and e bounded by the current radius of local *VG*, i.e., without triggering the retrieval of any new obstacle. Here, as proved in Lemma 4.1, $P_1(p, s)$ and $P_2(p, e)$ must represent the real shortest paths from p to s and e respectively, i.e., $P_1(p, s) = SP(p, s)$ and $P_2(p, e) = SP(p, e)$. In other words, the fact that IOR retrieves all the obstacles with their *mindist* to q not exceeding $\text{MAX}(1/2 \cdot (\text{MIN}(|P_1(p, s)|, |P_2(p, e)|) + |q| + mindist(p, q)), |P_1(p, s)|, |P_2(p, e)|)$ means that all the obstacles intersecting the search range $SR_{p,q}$ have been found, as proved in Theorem 4.1. Thus, the correctness of IOR algorithm is guaranteed.

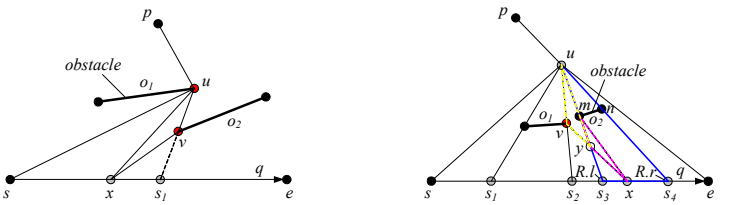
In addition, we would like to highlight that, since points in the data set P are accessed in ascending order of their *mindist* to q, IOR, for a data point $p \in P$, does not need to *start from scratch*. Specifically, the local visibility graph *VG* formed by p can be reused by another point p' that is visited after p. If p' does not trigger the retrieval of any new obstacle, i.e., current *VG* has already covered all the obstacles overlapping the search range $SR_{p',q}$, IOR, for the point p', can be safely terminated by reusing the current *VG*. Otherwise, it expands the local *VG* by adding new obstacle vertexes until all the obstacles that intersect $SR_{p',q}$ have been retrieved. Therefore, IOR is an *incremental* process, and it finds obstacles, for all the points in P, via *one* traversal of the obstacle set O.

4.2 Control Point List Computation

Once the local *VG* contains all the obstacles that may affect the obstructed distance from a specified data point p to q, our next step is to find out the control point list of p over q, i.e., $CPL_{p,q}$. Since the local *VG* includes all the obstacles affecting the obstructed distance between p and q, we can obtain $CPL_{p,q}$ using the local *VG*. A straightforward approach is to utilize the fact that a control point over an interval R must be *visible* to all points of R, and invoke Dijkstra's algorithm to compute the shortest path from p to every node (i.e., obstacle vertex) n that is within the current local *VG*. For each n located inside *VG* (i.e., $n \in VG$), we obtain n's visible region over q (i.e., $VR_{n,q}$), and add a new tuple $\langle n, R_n = VR_{n,q} \rangle$ to $CPL_{p,q}$ if $VR_{n,q} \neq \emptyset$, assuming that n is the control point of p over $VR_{n,q}$. If the interval R_n overlaps the interval R_m which is associated with some other control point m included in the current $CPL_{p,q}$, i.e., $\exists \langle m, R_m \rangle \in CPL_{p,q}$ with $R_n \cap R_m \neq \emptyset$, an update operation is triggered. Obviously, this method is expensive, especially when the number of nodes inside *VG* is large. To address this, we present several lemmas that can simplify the evaluation cost of some nodes $n \in VG$.

LEMMA 4.2. *Given a data point p, a query line segment q, and a node (i.e., an obstacle vertex) v in VG, we assume that the shortest path $SP(p, v)$ from p to v visits node u right before v. Let $VR_{u,q}$ and $VR_{v,q}$ be the visible regions of u and v over q, respectively. Point v cannot be the control point of p over any interval $R \subseteq (VR_{u,q} \cap VR_{v,q})$.*

PROOF. Pease refer to Appendix F. □



(a) v cannot be p's control point on $[s, s_1]$ (b) v cannot be p's control point on $[s_3, s_4]$

Fig. 8. Optimizations for control point list computation

As Dijkstra's algorithm gradually expands the search space from p (i.e., it always reaches u before v if $||p, u|| < ||p, v||$), Lemma 4.2 matches its traversal perfectly. Whenever a node v is examined, it must be reached by the shortest path from p, and hence the node u visited right before v along the path is known. As depicted in Figure 8(a), the shortest path from p to v passes u first and then reaches v. Instead of considering v's visible region over q, i.e., $VR_{v,q}$ ($= q = [s, e]$), we only need to consider the region that is not enclosed by $VR_{u,q}$, i.e., $VR_{v,q} - VR_{u,q}$ ($= [s, e] - [s, s_1] = [s_1, e]$). However, not all the intervals contained in ($VR_{v,q} - VR_{u,q}$) require evaluation. Lemma 4.3 can further shrink the search interval.

LEMMA 4.3. *Given a data point p, a query line segment q, and a node (i.e., an obstacle vertex) v in VG, we assume that the shortest path $SP(p, v)$ from p to v visits node u right before v, and $VR_{u,q}$ ($VR_{v,q}$) is the visible region of u (v) over q. Let $R = [R.l, R.r] \subseteq (VR_{v,q} - VR_{u,q})$ be the interval such that points along R are invisible to u due to the obstruction of obstacle o, and Δ be the triangle formed by points $R.l$, $R.r$, and u. Point v cannot become the control point of p over R if the following conditions are satisfied*: (*i*) *v is located outside the triangle Δ*, (*ii*) *all the line segments formed by v and any point on R intersect $[u, R.l]$ (or $[u, R.r]$), and* (*iii*) *point $R.l$ (or $R.r$) is blocked from u by a vertex v_o of o and v_o is visible to every point along R.*

PROOF. Pease refer to Appendix G. □

Take the case illustrated in Figure 8(b) as an example. For all the intervals included in $VR_{v,q} - VR_{u,q} = \{[s_1, s_2], [s_3, s_4]\}$, we can confirm that v cannot be p's control point over interval $[s_3, s_4]$ by Lemma 4.3. The above Lemmas 4.2 and 4.3 can be employed to reduce the examination cost of nodes included in the local VG. However, if the number of nodes contained in the local VG is huge, the examination cost is still high. Actually, not all the nodes in VG can change the current control point list. In order to further simplify the examination process, Lemma 4.4 is proposed.

LEMMA 4.4. *Given a data point p and a query line segment q, we assume that the current control point list of p over q, i.e., $CPL_{p,q}$, is $\cup_{i\in[1,m]} \{\langle cp_i, R_i\rangle\}$, and let $CPL_{MAX} = \mathrm{MAX}_{i\in[1,m]}(||p, cp_i|| + dist(cp_i, R_i.l), ||p, cp_i|| + dist(cp_i, R_i.r))$*[15]. *A node (i.e., an obstacle vertex) v in VG cannot be included in $CPL_{p,q}$ if $||p, v|| + mindist(v, q) \geq CPL_{MAX}$.*

PROOF. Pease refer to Appendix H. □

As an example, in Figure 9, p's control point list $CPL_{p,q}$ over q is $\{\langle p, [s, s_1]\rangle, \langle v_1, [s_1, s_2]\rangle, \langle v_2, [s_2, s_3]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_4, [s_4, e]\rangle\}$ and $CPL_{MAX} = dist(p, s)$. Since the node v_6 (i.e.,

[15] If $\exists \langle cp_i, R_i\rangle \in CPL_{p,q}$ with $cp_i = \varnothing$, $CPL_{MAX} = \infty$.

Algorithm 2 Control Point List Computation Algorithm (CPLC)

Input: a query line segment $q = [s, e]$; a data point p; a visibility graph VG
Output: p's control point list $CPL_{p,q}$ over q
/* Next(VG, p) is to expand the search space from p by Dijkstra's algorithm. */
1: $CPL_{p,q} = \{\langle \varnothing, [s, e]\rangle\}$
2: **while** v = Next(VG, p) ($\neq \varnothing$) **do**
3: **if** $\|p, v\| \geq CPL_{MAX}$ **then** // Lemma 4.4
4: **break**
5: let u be the node that $SP(p, v)$ passes right before reaching v
6: $R = VR_{v,q} - VR_{u,q}$ // Lemma 4.2
7: refine R using Lemma 4.3
8: **for** each tuple $\langle cp_i, R_i\rangle$ in $CPL_{p,q}$ **do** // update $CPL_{p,q}$
9: $R_{int} = R \cap R_i$
10: **if** $R_{int} \neq \varnothing$ AND $cp_i = \varnothing$ **then**
11: replace $\langle cp_i, R_i\rangle$ with $\langle v, R_{int}\rangle$ and $\langle cp_i, R_i - R_{int}\rangle$ (if $(R_i - R_{int}) \neq \varnothing$)
12: **else if** $R_{int} \neq \varnothing$ AND $cp_i \neq \varnothing$ **then**
13: $d = \|p, cp_i\| - \|p, v\|$
14: Split(p, cp_i, p, v, R_{int}, d) // see Section 3.2 for details
15: **return** $CPL_{p,q}$

a vertex of obstacle o_3) in VG satisfies $\|p, v_6\| + mindist(v_6, q) > dist(p, s) = CPL_{MAX}$, it is excluded from $CPL_{p,q}$ according to Lemma 4.4.

In fact, Lemma 4.4 serves as an *early termination condition* of the *Control Point List Computation Algorithm* (CPLC) whose pseudo-code is shown in Algorithm 2. CPLC shares the same basic idea as the straightforward approach mentioned at the beginning of Section 4.2. That is to call Dijkstra's algorithm to gradually traverse the local visibility graph VG and to access nodes v in VG according to ascending order of their obstructed distances to a specified data point p. The p's control point list $CPL_{p,q}$ over q is updated during the traversal. However, different from the simple method, CPLC utilizes Lemmas 4.2 and 4.3 (lines 6-7) to significantly reduce the node examination cost. The Split function invoked (line 14) is the same as the split point computation approach presented in Section 3.2. Before v is considered, all the shortest paths from p to any point on interval R_{int} ($\subseteq q$) pass the control point cp_i, and now the algorithm needs to check whether the path from p to any point along R_{int} via v is even shorter (lines 8-14). Suppose s' with coordinate $(x, 0)$ is a point on $R_{int} = [R_{int}.l, R_{int}.r] = R \cap R_i$. There are four cases, as discussed in Section 3.2, with $dist(v, s') - dist(cp_i, s') = d = \|p, cp_i\| - \|p, v\|$ and the difference between v's projection on R_{int} and cp_i's projection on R_{int} denoted by a. (i) **Case 1: $d \geq dist(cp_i, v)$.** $\langle cp_i, R_i\rangle$ is replaced with $\langle v, R_{int}\rangle$ and $\langle cp_i, R_i - R_{int}\rangle$ (if $R_i - R_{int} \neq \varnothing$), since $dist(v, s') - dist(cp_i, s') \leq d = \|p, cp_i\| - \|p, v\|$. (ii) **Case 2: $a < d < dist(cp_i, v)$.** Interval R_{int} will be decomposed into three sub-intervals by points x_1 and x_2 along R_{int}, with x_1 and x_2 derived based on Equation (1). Thereafter, $\langle cp_i, R_i\rangle$ is replaced accordingly. Specifically, $\langle cp_i, R_i\rangle$ is replaced with $\langle v, [R_{int}.l, x_1]\rangle$, $\langle cp_i, [x_1, x_2]\rangle$, $\langle v, [x_2, R_{int}.r]\rangle$, and $\langle cp_i, R_i - R_{int}\rangle$ (if $R_i - R_{int} \neq \varnothing$). (iii) **Case 3: $-a < d \leq a$.** Interval R_{int} will be decomposed into two sub-intervals by point x_1 on R_{int}, with x_1 derived based on Equation (1) too. Again, $\langle cp_i, R_i\rangle$ is replaced accordingly. In particular, $\langle cp_i, R_i\rangle$ is replaced with $\langle cp_i, [R_{int}.l, x_1]\rangle$, $\langle v, [x_1, R_{int}.r]\rangle$, and $\langle cp_i, R_i - R_{int}\rangle$ (if $R_i - R_{int} \neq \varnothing$). (iv) **Case 4: $d \leq -a$.** $\langle cp_i, R_i\rangle$ is still valid.

The above process proceeds until all the nodes in local VG are accessed or the visited node satisfies $\|p, v\| \geq CPL_{MAX}$ (lines 3-4). As nodes in the local VG are traversed in

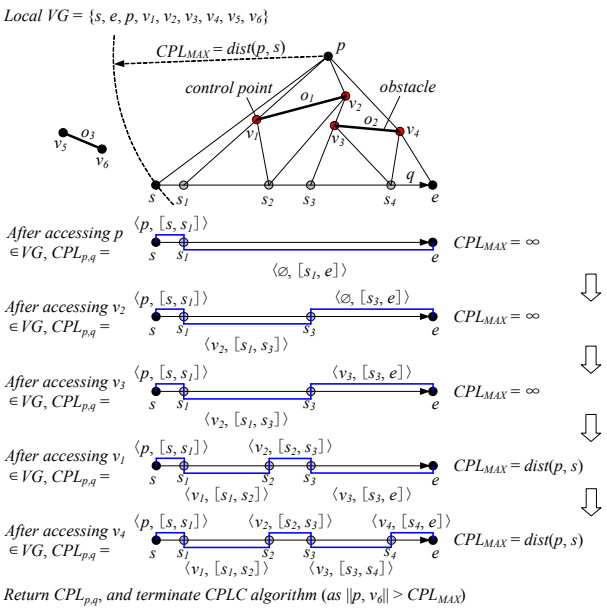


Fig. 9. Example of CPLC algorithm

ascending order of their obstructed distances to p, all the remaining nodes n in VG must satisfy $\|p, n\| \geq CPL_{MAX}$ when the currently visited node has its obstructed distance to p larger than CPL_{MAX}. Note that the termination condition employed in CPLC relaxes Lemma 4.4 using *zero* as the lower bound of $mindist(n, q)$ in order to reduce the distance computation overhead.

We illustrate the CPLC algorithm with the example depicted in Figure 9, where the local $VG = \{s, e, p, v_1, v_2, v_3, v_4, v_5, v_6\}$. First, CPLC accesses node $p \in VG$. As p is indeed the data point evaluated, $R = VR_{p,q} = \{[s, s_1]\}$ and $CPL_{p,q}$ is updated to $\{\langle p, [s, s_1]\rangle, \langle \emptyset, [s_1, e]\rangle\}$. Next, CPLC visits node $v_2 \in VG$. Since the shortest path $SP(p, v_2)$ from p to v_2 does not pass through any node u in VG (i.e., $u = \emptyset$), $R = VR_{v2,q} = \{[s, s_3]\}$ and $CPL_{p,q}$ is changed to $\{\langle p, [s, s_1]\rangle, \langle v_2, [s_1, s_3]\rangle, \langle \emptyset, [s_3, e]\rangle\}$. Then, node $v_3 \in VG$ is accessed. As the shortest path $SP(p, v_3)$ from p to v_3 passes node v_2 first and then v_3, $R = VR_{v3,q} - VR_{v2,q} = \{[s_3, e]\}$. Note that, here CPLC cannot refine R using Lemma 4.3 because v_3 is located inside the triangle v_2s_3E. After updating $CPL_{p,q}$, $CPL_{p,q}$ becomes $\{\langle p, [s, s_1]\rangle, \langle v_2, [s_1, s_3]\rangle, \langle v_3, [s_3, e]\rangle\}$. Similarly, the subsequent nodes in VG visited are v_1, v_4 (in this order), after which $CPL_{p,q} = \{\langle p, [s, s_1]\rangle, \langle v_1, [s_1, s_2]\rangle, \langle v_2, [s_2, s_3]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_4, [s_4, e]\rangle\}$. Finally, CPLC accesses $v_6 \in VG$. Since $\|p, v_6\| > CPL_{MAX} = dist(p, s)$, CPLC outputs $CPL_{p,q}$ and terminates. For ease of understanding, Figure 10 shows the trace of CPLC algorithm for this running example.

4.3 Result List Update

Once a new data point p is accessed and p's control point list over a given query line segment q (i.e., $CPL_{p,q}$) is formed, the next step is to evaluate the impact of p on current result list RL. The basic idea is to check whether p will replace the current ONN for some points along q. The pseudo-code of the *Result List Update Algorithm* (RLU) is presented

Algorithm 3 Result List Update Algorithm (RLU)

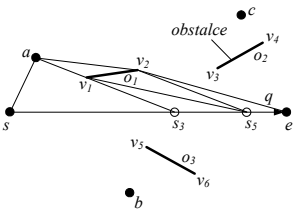
Input: a data point p; p's control point list $CPL_{p,q}$ over q; current result list RL
Output: the updated result list

```
 1: for each tuple ⟨p_i, cp_i, R_i⟩ ∈ RL do
 2:   for each tuple ⟨cp_i', R_i'⟩ ∈ CPL_{p,q} do
 3:     if R_i ∩ R_i' ≠ ∅ then
 4:       R_int = R_i ∩ R_i' = [l, r]
 5:       R_dif = R_i − R_int
 6:       R_dif' = R_i' − R_int
 7:       if R_dif ≠ ∅ then
 8:         add ⟨p_i, cp_i, R_dif⟩ to RL
 9:       if R_dif' ≠ ∅ then
10:         add ⟨cp_i', R_dif'⟩ to CPL_{p,q}
11:       if ||p, cp_i'|| + dist(cp_i', l) > ||p_i, cp_i|| + dist(cp_i, l) AND ||p, cp_i'|| + dist(cp_i', r) >
          ||p_i, cp_i|| + dist(cp_i, r) AND dist_⊥(cp_i', R_int) > dist_⊥(cp_i, R_int) then   // Lemma 3.1
12:         add ⟨p_i, cp_i, R_int⟩ to TRL and Merge() if necessary
13:         continue
14:       else
15:         d = ||p_i, cp_i|| − ||p, cp_i'||
16:         Split(p_i, cp_i, p, cp_i', R_int, d)   // see Section 3.2 for details
17:         insert the result tuples from Split function into TRL and Merge() if necessary
18: return TRL
```

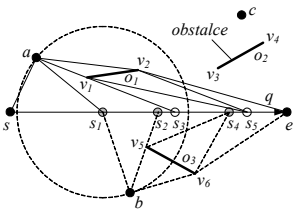
in Algorithm 3, which incrementally updates RL.

RLU performs the update by scanning the current result list RL. For each tuple $\langle p_i, cp_i, R_i\rangle$ in RL, it finds the corresponding tuples $\langle cp_i', R_i'\rangle$ in $CPL_{p,q}$ with $R_i \cap R_i' \neq \emptyset$. For such a tuple, RLU first derives the intersection R_{int} ($= R_i \cap R_i' = [l, r]$) and differences R_{dif} ($= R_i - R_{int}$), R_{dif}' ($= R_i' - R_{int}$) between R_i and R_i' (lines 4-6). Thereafter, RLU distinguishes four cases: (i) if R_{dif} is not empty (i.e., $R_{dif} \neq \emptyset$), $\langle p_i, cp_i, R_{dif}\rangle$ is inserted into RL for further evaluation (lines 7-8); (ii) if R_{dif}' is not empty (i.e., $R_{dif}' \neq \emptyset$), $\langle cp_i', R_{dif}'\rangle$ is added to $CPL_{p,q}$ (lines 9-10); (iii) if $\|p, cp_i'\| + dist(cp_i', l) > \|p_i, cp_i\| + dist(cp_i, l)$, $\|p, cp_i'\| + dist(cp_i', r) > |p_i, cp_i\| + dist(cp_i, r)$, and $dist_\perp(cp_i', R_{int}) > dist_\perp(cp_i, R_{int})$, p_i is closer to any point along the interval $[l, r]$ than p by Lemma 3.1 (presented in Section 3.2), and hence $\langle p_i, cp_i, R_{int}\rangle$ is inserted into a temporal result list TRL (lines 11-13); and otherwise (iv) RLU utilizes the split point computation approach (i.e., Split function) discussed in Section 3.2 to update RL, and adds the result tuples to TRL (lines 14-17). After all the tuples in RL are evaluated, RLU outputs TRL as the updated result list (line 18). It is important to note that, when a new tuple $\langle p', cp', R'\rangle$ in the current RL is inserted into TRL, it might be merged with an existing tuple $\langle p'', cp'', R''\rangle$ in TRL if $cp' = cp''$ and intervals R', R'' are *adjacent*, with the merge operation represented by Merge().

Figure 10 depicts an example with the data set $P = \{a, b, c\}$, the obstacle set $O = \{o_1, o_2, o_3\}$, and a specified query line segment $q = [s, e]$. Suppose point a has been processed and current result list $RL = \{\langle a, a, [s, s_3]\rangle, \langle a, v_1, [s_3, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$, as shown in Figure 10(a). Now we invoke RLU to evaluate the impact of a new data point b on RL, with b's control point list $CPL_{b,q}$ over q being $\{\langle b, [s, s_2]\rangle, \langle v_5, [s_2, s_4]\rangle, \langle v_6, [s_4, e]\rangle\}$. RLU recursively examines every tuple in RL. First, the tuple $\langle a, a, [s, s_3]\rangle \in RL$ is compared against $\langle b, [s, s_2]\rangle \in CPL_{b,q}$. As $[s, s_3] \cap [s, s_2] = [s, s_2] \neq \emptyset$, RLU derives R_{int} ($= [s, s_2]$), R_{dif} ($= [s, s_3] - [s, s_2] = [s_2, s_3] \neq \emptyset$), and R_{dif}' ($= [s, s_2] - [s, s_2] = \emptyset$). It adds $\langle a, a, [s_2, s_3]\rangle$



(a) Before updating current *RL*



(b) After updating current *RL*

R_i	R_i'	R_{int}	R_{dif}	R_{dif}'	*RL*	$CPL_{b,q}$	*TRL*
$[s, s_3]$	$[s, s_2]$	$[s, s_2]$	$[s_2, s_3]$	$\varnothing$	$\{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_3, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$	$\{\langle b, [s, s_2]\rangle, \langle v_3, [s_2, s_4]\rangle, \langle v_6, [s_4, e]\rangle\}$	$\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle\}$
$[s_2, s_3]$	$[s_2, s_4]$	$[s_2, s_3]$	$\varnothing$	$[s_3, s_4]$	$\{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_3, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$	$\{\langle b, [s, s_2]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_6, [s_4, e]\rangle\}$	$\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_3, [s_2, s_3]\rangle\}$
$[s_3, s_5]$	$[s_3, s_4]$	$[s_3, s_4]$	$[s_4, s_5]$	$\varnothing$	$\{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_4, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$	$\{\langle b, [s, s_2]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_6, [s_4, e]\rangle\}$	$\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_3, [s_2, s_4]\rangle\}$
$[s_4, s_5]$	$[s_4, e]$	$[s_4, s_5]$	$\varnothing$	$[s_5, e]$	$\{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_4, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$	$\{\langle b, [s, s_2]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_6, [s_5, e]\rangle\}$	$\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_3, [s_2, s_4]\rangle, \langle b, v_6, [s_4, s_5]\rangle\}$
$[s_5, e]$	$[s_5, e]$	$[s_5, e]$	$\varnothing$	$\varnothing$	$\{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_4, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$	$\{\langle b, [s, s_2]\rangle, \langle v_3, [s_3, s_4]\rangle, \langle v_6, [s_5, e]\rangle\}$	$\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_3, [s_2, s_4]\rangle, \langle b, v_6, [s_4, e]\rangle\}$

Return TRL, and terminate RLU algorithm

(c) The trace of RLU algorithm for the running example

Fig. 10. Example of RLU algorithm

to *RL*, and partitions $[s, s_2]$ into two sub-intervals (i.e., Case 3 presented in Section 4.2) based on the Split function, after which $RL = \{\langle a, a, [s_2, s_3]\rangle, \langle a, v_1, [s_3, s_5]\rangle, \langle a, v_2, [s_5, e]\rangle\}$ and $TRL = \{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle\}$. Next, the tuple $\langle a, a, [s_2, s_3]\rangle \in RL$ is compared with $\langle v_5, [s_2, s_4]\rangle \in CPL_{b,q}$. Again, since $[s_2, s_3] \cap [s_2, s_4] = [s_2, s_3] \neq \varnothing$, RLU derives R_{int} ($= [s_2, s_3]$), R_{dif} ($= [s_2, s_3] - [s_2, s_3] = \varnothing$), and R_{dif}' ($= [s_2, s_4] - [s_2, s_3] = [s_3, s_4]$). It inserts $\langle v_5, [s_3, s_4]\rangle$ to $CPL_{p,q}$, and updates *TRL* to $\{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_5, [s_2, s_3]\rangle\}$ using the Split function. The process proceeds in the same manner until all the tuples in *RL* are evaluated. Finally, as illustrated in Figure 10(b), RLU returns $TRL = \{\langle a, a, [s, s_1]\rangle, \langle b, b, [s_1, s_2]\rangle, \langle b, v_5, [s_2, s_4]\rangle, \langle b, v_6, [s_4, e]\rangle\}$ as the updated *RL*. To facilitate understanding, Figure 10(c) summarizes the detailed steps of RLU algorithm with respect to this running example.

4.4 CONN Search Algorithm

Having explained IOR, CPLC, and RLU, we are ready to present the complete CONN query processing algorithm, namely *CONN Search Algorithm* (CONN). Algorithm 4 shows the pseudo-code of CONN. It takes an R-tree T_p on the data set *P*, an R-tree T_o on

Algorithm 4 CONN Search Algorithm (CONN)

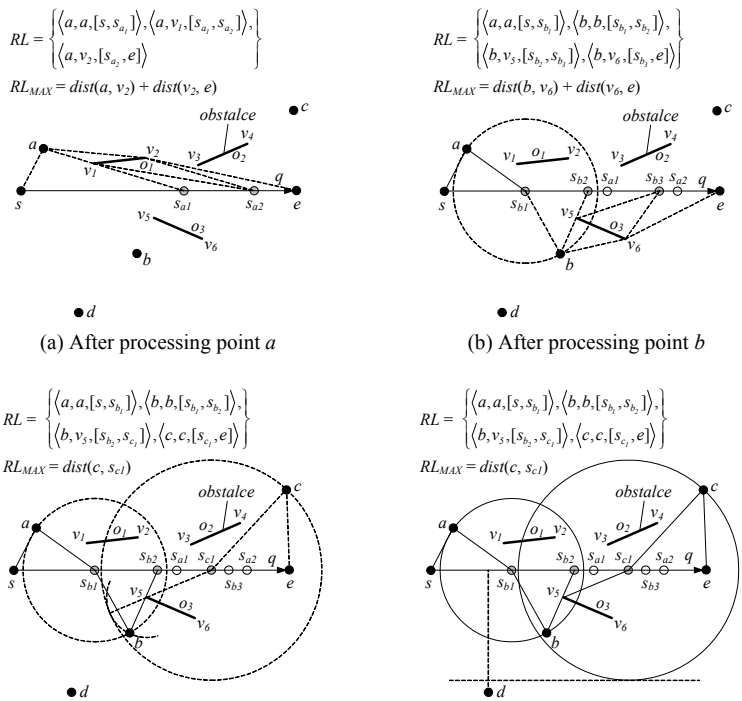
Input: a data R-tree T_p; an obstacle R-tree T_o; a query line segment $q = [s, e]$
Output: the result list RL of a CONN query

```
1:  RL = {⟨∅, ∅, [s, e]⟩}, RL_MAX = ∞, VG = {s, e}, and d = 0
2:  insert (root(T_p), 0) and (root(T_o), 0) into heaps H_p and H_o, respectively
3:  while H_p ≠ ∅ do
4:     de-heap the top entry (e, key) of H_p
5:     if key ≥ RL_MAX then   // Lemma 3.2, key = mindist(e, q)
6:        break
7:     else if e is a data point then
8:        insert e into local visibility graph VG
9:        IOR (T_o, H_o, q, e, VG, d)   // Algorithm 1
10:       CPL_e,q = CPLC (q, e, VG)   // Algorithm 2
11:       remove e from VG
12:       RL = RLU (e, CPL_e,q, RL)   // Algorithm 3
13:    else   // e is an intermediate (i.e., a non-leaf) node
14:       for each child entry e_i ∈ e do
15:          insert (e_i, mindist(e_i, q)) into H_p
16: return RL
```

the obstacle set O, and a query line segment q as input, and outputs the final result list RL for a CONN query.

CONN follows the *best-first* traversal. For this purpose, 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 Euclidean distances (i.e., *mindist*) to q. Initially, CONN inserts the root nodes of T_p and T_o into H_p and H_o, respectively (line 2). Thereafter, it continuously de-heaps the head entry e of H_p for examination until H_p becomes empty (lines 3-15). Each examination involves two tasks. First, CONN checks the *early termination condition* (presented in Lemma 3.2), and terminates if $mindist(e, q) \geq RL_{MAX}$ because all the unexamined data points in P cannot change the current result list RL obtained, as proved in Lemma 3.2 (lines 5-6). Otherwise, CONN proceeds with the second task to evaluate the top entry e of H_p. If e is a data point, CONN inserts e into local visibility graph VG, invokes IOR algorithm to retrieve all the obstacles that may affect the obstructed distances from e to any point along q, calls CPLC algorithm to get e's control point list $CPL_{e,q}$ over q, removes e from VG, and utilizes RLU algorithm to update the current result list RL (lines 7-12). On the other hand, e is an intermediate (i.e., a non-leaf) node. CONN inserts all the child entries of e into H_p (lines 13-15).

To facilitate the understanding, we illustrate the CONN algorithm using an example depicted in Figure 11, with data set $P = \{a, b, c, d\}$, obstacle set $O = \{o_1, o_2, o_3\}$, and a query line segment $q = [s, e]$. First of all, the result list RL is initialized to $\{\langle \emptyset, \emptyset, [s, e]\rangle\}$. When the first data point a (that is the closest to q in the Euclidean space) is visited, CONN inserts the point a into VG (= $\{s, e, a\}$), and calls IOR to retrieve all the obstacles that may affect the obstructed distances from a to any point along q, i.e., $S_o = \{o_1, o_2, o_3\}$ and VG (= $\{s, e, a, v_1, v_2, v_3, v_4, v_5, v_6\}$. Then, it employs CPLC to obtain $CPL_{a,q} = \{\langle a, a, [s, s_{a1}]\rangle, \langle a, v_1, [s_{a1}, s_{a2}]\rangle, \langle a, v_2, [s_{a2}, e]\rangle\}$, i.e., the control point list of a over q. Next, point a is deleted from VG, and then RLU is invoked to update the current RL, after which $RL = \{\langle a, a, [s, s_{a1}]\rangle, \langle a, v_1, [s_{a1}, s_{a2}]\rangle, \langle a, v_2, [s_{a2}, e]\rangle\}$ as shown in Figure 11(a). Similarly, the second data point processed is b, and Figure 11(b) depicts the



(a) After processing point *a* (b) After processing point *b*

(c) After processing point *c* (d) After processing point *d*

Fig. 11. Illustration of CONN query processing

corresponding $RL = \{\langle a, a, [s, s_{b1}]\rangle, \langle b, b, [s_{b1}, s_{b2}]\rangle, \langle b, v_5, [s_{b2}, s_{b3}]\rangle, \langle b, v_6, [s_{b3}, e]\rangle\}$ after the processing of b. Subsequently, CONN evaluates the third data point c and updates RL again to $\{\langle a, a, [s, s_{b1}]\rangle, \langle b, b, [s_{b1}, s_{b2}]\rangle, \langle b, v_5, [s_{b2}, s_{c1}]\rangle, \langle c, c, [s_{c1}, e]\rangle\}$, which is illustrated in Figure 11(c). Finally, when the last data point d is encountered, it is discarded directly as $mindist(d, q) > RL_{MAX} = dist(c, s_{c1})$. Here, the algorithm terminates, with the final query result $RL = \{\langle a, a, [s, s_{b1}]\rangle, \langle b, b, [s_{b1}, s_{b2}]\rangle, \langle b, v_5, [s_{b2}, s_{c1}]\rangle, \langle c, c, [s_{c1}, e]\rangle\}$ (i.e., $\{\langle a, [s, s_{b1}]\rangle, \langle b, [s_{b1}, s_{c1}]\rangle, \langle c, [s_{c1}, e]\rangle\}$ for short), as shown in Figure 11(d).

Next, we analyze some properties of the CONN algorithm and prove its correctness.

LEMMA 4.5. *The CONN algorithm only processes the data points and obstacles* relevant to the final query result.

PROOF. Pease refer to Appendix I. □

LEMMA 4.6. *The CONN algorithm traverses the data R-tree T_p and the obstacle R-tree T_o* at most once.

PROOF. Pease refer to Appendix J. □

THEOREM 4.2. *The CONN algorithm retrieves* exactly *the ONN of each point along a given query line segment*, *i.e.*, *the CONN algorithm has* no false misses *and* no false hits.

PROOF. Pease refer to Appendix K. □

4.5 Discussion

Our previously proposed CONN algorithm assumes the data set P and the obstacle set O are indexed by two *separate* R-trees. However, it can be naturally extended to answer CONN search based on a *single* R-tree that indexes both P and O.

The detailed extensions are summarized as follows: (i) It requires only one heap H to store candidate entries (including data points, obstacles, and non-leaf nodes), sorted in ascending order of their minimum Euclidean distances (i.e., *mindist*) to a given query line segment q. (ii) When processing the top entry e de-heaped from H, it distinguishes three cases. (1) Case 1: e is an obstacle. It inserts e's vertexes into the local visibility graph VG. (2) Case 2: e is a data point. It calls IOR to retrieve all the obstacles that may affect the obstructed distances from e to any point along q, invokes CPLC to get e's control point list $CPL_{e,q}$ over q, and employs RLU to update the current result list RL. It is worth noting that, during the obstacle retrieval via IOR, it is possible to access some data points which will be en-heaped into H for later processing. (3) Case 3: e is a non-leaf node. All the child entries of e are inserted into H for later evaluation.

5. VARIATIONS OF CONN QUERIES

In this section, we demonstrate the flexibility and extensibility of the proposed CONN algorithm by studying two interesting variations of CONN queries, namely, *continuous obstructed k nearest neighbor* (CO*k*NN) search and *trajectory obstructed nearest neighbor* (TONN) search. Note that TONN retrieval can be generalized to *trajectory obstructed k nearest neighbor* (TO*k*NN) search. As the search algorithm for TONN can be extended to that for TO*k*NN naturally, which is similar to the extension from CONN retrieval to CO*k*NN search, we omit its detailed explanation.

5.1 CO*k*NN Search

CO*k*NN search aims at finding the k (≥ 1) ONNs for every point along a specified query line segment q. The proposed algorithms for CONN queries can be extended to support CO*k*NN retrieval. The detailed extensions are described as follows.

First, ONN is generalized to obstructed k nearest neighbors (O*k*NNs), as defined in Definition 5.1, based on which CO*k*NN search is formulated in Definition 5.2. Note that, although the CO*k*NN query defined in Definition 5.2 refers to *ordered k*NN search, the CONN query studied in this article can also be adapted to support *orderless* CO*k*NN retrieval via merging query result.

Definition 5.1 (*Obstructed k Nearest Neighbors*). Given $p \in P$ and a query point q', p is one of the *obstructed k nearest neighbors* (O*k*NNs) of q' iff there are at most $(k-1)$ data points $p' \in P - \{p\}$ that have their obstructed distances to q' smaller than the obstructed distance between p and q', i.e., $|\{p' \in P - \{p\} \mid \|p, q'\| > \|p', q'\|\}| \leq (k-1)$.

Definition 5.2 (*Continuous Obstructed k Nearest Neighbor Query*). Given P, O, and q, a *continuous obstructed k nearest neighbor* (CO*k*NN) *query* returns the result list RL that contains a set of $\langle ONNS_i, R_i\rangle$ ($i \in [1, t]$) tuples, such that (i) $\cup_{i\in[1,t]} R_i = q$; (ii) $\forall\ i, j \in [1, t]$ ($i \neq j$ and $|i-j| \neq 1$), $R_i \cap R_j = \varnothing$; (iii) $\forall\ i \in [1, t)$, R_i (= $[R_i.l, R_i.r]$) $\cap$ R_{i+1} (= $[R_{i+1}.l, R_{i+1}.r]$) = $R_i.r$; (iv) $\forall\ i \in [1, t]$, $|ONNS_i| = k$ (in this article we assume $|P| \geq k$); and (v) $\forall\ \langle ONNS_i, R_i\rangle \in RL$, $ONNS_i$ is the set of O*k*NNs for every point on interval R_i.

As mentioned in Section 3.2, to facilitate the split point computation under obstacle constraints, we introduce the concept of *control point* (defined in Definition 3.9). Accordingly, the result list RL of CO*k*NN retrieval is reformatted into the list of *three-element* tuples $\langle ONNS_i, CPS_i, R_i\rangle$ ($i \in [1, t]$), with $ONNS_i$ representing the set of O*k*NNs

for every point along interval R_i and CPS_i being the set of control points. To be more specific, $ONNS_i = \cup_{j\in[1,\,k]}\, ONN_j$, and $CPS_i = \cup_{j\in[1,\,k]}\, \langle CP_j, R_i\rangle$ in which CP_j is the control point of ONN_j over R_i, i.e., the shortest path from the answer object ONN_j to any point on the interval R_i must pass the control point CP_j, and CP_j is visible to any point along R_i.

Second, the pruning distance RL_{MAX} introduced in Lemma 3.2 needs to be updated, as presented in Lemma 5.1.

LEMMA 5.1. *Suppose the current result list* $RL = \cup_{i\in[1,\,t]}\, \langle ONNS_i, CPS_i, R_i\rangle$ *with interval* $R_i = [R_i.l, R_i.r] \subseteq q$. *Given a data point p and a query line segment* $q = [s, e]$, *p cannot change RL if* $mindist(p, q) > RL_{MAX} = \mathrm{MAX}_{i\in[1,\,t]}(maxodist(ONNS_i, R_i.l), maxodist(ONNS_i, R_i.r))$, *with* $maxodist(ONNS, s')$ *defined as follows*:

$$maxodist(ONNS, s') = \begin{cases} \mathrm{MAX}_{\forall p\in ONNS} \left\| p, s' \right\| & if \left|ONNS\right| = k \\ \infty & if \left|ONNS\right| < k \end{cases} \tag{4}$$

PROOF. Pease refer to Appendix L. □

Third, the handling of data points is similar as that for CONN search. Specifically, the processing of each data point $p \in P$ involves three steps. The first step is to find all the obstacles that may affect the obstructed distances from p to any point on q. The second step is to obtain the control point list of p over q (i.e., $CPL_{p,q}$). Finally, the third step is to update the current result list RL retrieved if necessary, where split point computation is more complex than that under CONN retrieval. Algorithm 5 presents the pseudo-code of the *Split Point Computation for COkNN Search Algorithm* (SPC-CO*k*NN), which evaluates the impact of a new data point on a specified interval R.

SPC-CO*k*NN takes as inputs a temporary result list TRL, an interval $R = [l, r]$, a set S_{oknn} that stores the current k ONNs for R retrieved so far, a set S_{kcp} that keeps the current k control points identified so far corresponding to points in S_{oknn} over R, a new data point p, and p's control point cp, and outputs the updated TRL. If R is empty (i.e., $R = \emptyset$), SPC-CO*k*NN returns TRL and terminates (lines 2-3). Otherwise, the algorithm performs the following tasks. First, SPC-CO*k*NN utilizes, for each control point cp_i in S_{kcp}, the split point computation approach (i.e., Split function) proposed in Section 3.2 to compute the split points corresponding to cp and cp_i on R, and preserves the result tuples from Split function in a set S_{spc} (lines 5-8). Note that, the set S_{spc} accepts entries/tuples in the form of $\langle cp', R'\rangle$, where $R' \subseteq R$ is the (sub) interval bounded by two consecutive split points, denoted as $R' = [R'.l, R'.r]$, and cp' is the corresponding control point for R'.

Then, SPC-CO*k*NN finds all the tuples $\langle cp', R'\rangle$ in S_{spc} such that the interval R' starts from l, and stores them in a set S_{spc}', i.e., $S_{spc}' = \{\langle cp', R'\rangle \mid \langle cp', R'\rangle \in S_{spc} \wedge R'.l = l\}$ (line 9). In the sequel, SPC-CO*k*NN distinguishes two cases (lines 10-24). (i) Case 1: all the tuples $\langle cp', R'\rangle$ in S_{spc}' have their corresponding control point cp' being cp_i but not cp (i.e., $\forall\ \langle cp', R'\rangle \in S_{spc}'$, $cp' \neq cp$). It identifies from S_{spc}' the interval $R_{min} \subseteq R$ having the shortest length (i.e., $R_{min} = \mathrm{MIN}(\{R' \mid \langle cp', R'\rangle \in S_{spc}'\})$), inserts tuple $\langle S_{oknn}, S_{kcp}, R_{min}\rangle$ into TRL (as both S_{oknn} and S_{kcp} remain valid for the interval R_{min}), and invokes recursively SPC-CO*k*NN to check the validity of S_{oknn} on the interval $R'' = R - R_{min}$ (if $R'' \neq \emptyset$) upon the presence of p (lines 10-15). (ii) Case 2: there is at least one tuple $\langle cp', R'\rangle$ of S_{spc}' with $cp' = cp$. It first finds from S_{spc}' the interval $R_{max} \subseteq R$ that has the longest length and meanwhile cp is the corresponding control point over R_{max}, i.e., $R_{max} = \mathrm{MAX}(\{R' \mid \langle cp', R'\rangle \in S_{spc}' \wedge cp' = cp\})$ (line 17). Suppose $R_{max} = [l, R_{max}.r]$, cp'' is the current control point in S_{kcp} such that cp and cp'' can generate the split point $R_{max}.r$ using the Split

Algorithm 5 Split Point Computation for CO*k*NN Search Algorithm (SPC-CO*k*NN)

Input: a temporary result list *TRL*; an interval/region $R = [l, r]$; a set $S_{oknn} = \{p_i \mid i \in [1, k]\}$ of the current k ONNs for every point along R; a set $S_{kcp} = \{cp_i \mid i \in [1, k]\}$ of the current k control points corresponding to S_{oknn} over R; a data point p; a control point cp

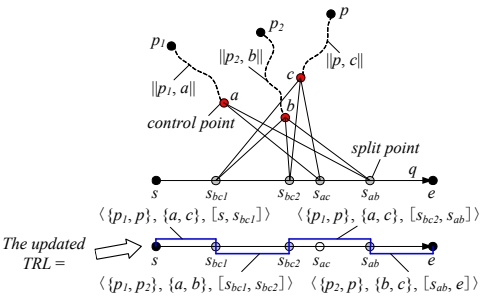
Output: the updated *TRL*

1: initialize a set $S_{spc} = \emptyset$ accepting entries/tuples in the form $\langle cp', R' \rangle$ with $R' = [R'.l, R'.r]$
2: **if** $R = \emptyset$ **then**
3: **return** *TRL*
4: **else** // R is not empty
5: **for** each control point $cp_i \in S_{kcp}$ **do**
6: $d = \|p_i, cp_i\| - \|p, cp\|$
7: Split(p_i, cp_i, p, cp, R, d) // compute split points
8: insert the result tuples from Split function into S_{spc}
9: $S_{spc}' = \{\langle cp', R' \rangle \mid \langle cp', R' \rangle \in S_{spc} \wedge R'.l = l\}$
10: **if** $cp' \neq cp$ for $\forall \langle cp', R' \rangle \in S_{spc}'$ **then**
11: $R_{min} = \text{MIN}(\{R' \mid \langle cp', R' \rangle \in S_{spc}'\})$
12: $R'' = R - R_{min}$
13: insert $\langle S_{oknn}, S_{kcp}, R_{min} \rangle$ into *TRL* // both S_{oknn} and S_{kcp} remain valid for the interval R_{min}
14: **if** $R'' \neq \emptyset$ **then**
15: SPC-CO*k*NN (*TRL*, R'', S_{oknn}, S_{kcp}, p, cp) // split the interval R'' if necessary
16: **else** // $\exists\ cp' = cp$ for $\forall \langle cp', R' \rangle \in S_{spc}'$
17: $R_{max} = \text{MAX}(\{R' \mid \langle cp', R' \rangle \in S_{spc}' \wedge cp' = cp\})$ // $R_{max} = [l, R_{max}.r]$
18: let cp'' be the current control point in S_{kcp} such that cp and cp'' can generate the split point $R_{max}.r$, and p'' be the current ONN in S_{oknn} for the interval R_{max} corresponding to cp''
19: $R'' = R - R_{max}$
20: $S_{oknn}' = S_{oknn} - p'' \cup p$ // replace p'' with p
21: $S_{kcp}' = S_{kcp} - cp'' \cup cp$ // replace cp'' with cp
22: SPC-CO*k*NN (*TRL*, R_{max}, S_{oknn}', S_{kcp}', p'', cp'')
23: **if** $R'' \neq \emptyset$ **then**
24: SPC-CO*k*NN (*TRL*, R'', S_{oknn}, S_{kcp}, p, cp)
25: **return** *TRL*

function, and p'' is the current ONN in S_{oknn} for the interval R_{max} that corresponds to the control point cp'' (line 18). The algorithm then replaces cp'' and p'' with cp and p respectively, employs SPC-CO*k*NN to examine the validity of $(S_{oknn} - p'' \cup p)$ on the interval R_{max} upon the existence of p'', and calls recursively SPC-CO*k*NN to check the validity of S_{oknn} on the interval $R'' = R - R_{max}$ (if $R'' \neq \emptyset$) upon the presence of p (lines 19-24). Finally, the updated *TRL* is returned to complete the algorithm (line 25).

We use the example depicted in Figure 12 to illustrate the SPC-CO*k*NN algorithm. Suppose a CO*2*NN ($k = 2$) query is issued at a specified query line segment $q = [s, e]$. We assume that points a and b are the current control points of data points p_1 and p_2 over q respectively, and SPC(cp_1, cp_2, R) represents the result of the split point computation between two control points cp_1, cp_2 over the interval/region R via Split function, which contains a set of $\langle cp, R' \rangle$ tuples such that cp is the control point for the (sub) interval $R' \subseteq R$. Now we employ SPC-CO*k*NN to evaluate the impact of a new control point c of a data point p over q, i.e., whether c will generate some new split points and become one of control points over some (sub) intervals on q.

In the first place, as $q = [s, e] \neq \emptyset$, SPC($a$, c, q) = $\{\langle a, [s, s_{ac}] \rangle, \langle c, [s_{ac}, e] \rangle\}$ and SPC(b, c, q) = $\{\langle c, [s, s_{bc1}] \rangle, \langle b, [s_{bc1}, s_{bc2}] \rangle, \langle c, [s_{bc2}, e] \rangle\}$ are computed by using the Split function,



(a) After updating *TRL*

S_{spc}'	R_{min}	R_{max}	cp''	p''	operation	*TRL*
$\{\langle a,[s,s_{ac}]\rangle, \langle c,[s,s_{bc1}]\rangle\}$	–	$[s, s_{bc1}]$	b	p_2	SPC-CO*k*NN (*TRL*, $[s, s_{bc1}]$, $\{p_1, p\}$, $\{a, c\}$, p_2, b) SPC-CO*k*NN (*TRL*, $[s_{bc1}, e]$, $\{p_1, p_2\}$, $\{a, b\}$, p, c)	$\varnothing$
$\{\langle a,[s,s_{bc1}]\rangle, \langle c,[s,s_{bc1}]\rangle\}$	$[s, s_{bc1}]$	–	–	–	–	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle\}$
$\{\langle a,[s,s_{bc1}],s_{ac}]\rangle, \langle b,[s_{bc1},s_{bc2}]\rangle\}$	$[s_{bc1}, s_{bc2}]$	–	–	–	SPC-CO*k*NN (*TRL*, $[s_{bc2}, e]$, $\{p_1, p_2\}$, $\{a, b\}$, p, c)	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle, \langle\{p_1, p_2\},\{a,b\},[s_{bc1},s_{bc2}]\rangle\}$
$\{\langle a,[s_{bc2},s_{ac}]\rangle, \langle c,[s_{bc2},e]\rangle\}$	–	$[s_{bc2}, e]$	b	p_2	SPC-CO*k*NN (*TRL*, $[s_{bc2}, e]$, $\{p_1, p\}$, $\{a, c\}$, p_2, b)	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle, \langle\{p_1, p_2\},\{a,b\},[s_{bc1},s_{bc2}]\rangle\}$
$\{\langle a,[s_{bc2},s_{ab}]\rangle, \langle c,[s_{bc2},e]\rangle\}$	$[s_{bc2}, s_{ab}]$	–	–	–	SPC-CO*k*NN (*TRL*, $[s_{ab}, e]$, $\{p_1, p\}$, $\{a, c\}$, p_2, b)	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle, \langle\{p_1, p_2\},\{a,b\},[s_{bc1},s_{bc2}]\rangle, \langle\{p_1, p\},\{a,c\},[s_{bc2},s_{ab}]\rangle\}$
$\{\langle b,[s_{ab},e]\rangle, \langle c,[s_{ab},e]\rangle\}$	–	$[s_{ab}, e]$	a	p_1	SPC-CO*k*NN (*TRL*, $[s_{ab}, e]$, $\{p, p_2\}$, $\{b, c\}$, p_1, a)	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle, \langle\{p_1, p_2\},\{a,b\},[s_{bc1},s_{bc2}]\rangle, \langle\{p_1, p\},\{a,c\},[s_{bc2},s_{ab}]\rangle\}$
$\{\langle b,[s_{ab},e]\rangle, \langle c,[s_{ab},e]\rangle\}$	$[s_{ab}, e]$	–	–	–	–	$\{\langle\{p_1, p\},\{a,c\},[s,s_{bc1}]\rangle, \langle\{p_1, p_2\},\{a,b\},[s_{bc1},s_{bc2}]\rangle, \langle\{p_1, p\},\{a,c\},[s_{bc2},s_{ab}]\rangle, \langle\{p_2, p\},\{b,c\},[s_{ab},e]\rangle\}$

Return TRL, and terminate SPC-COkNN algorithm

(b) The trace of SPC-CO*k*NN algorithm for the running example

Fig. 12. Example of SPC-CO*k*NN algorithm

and $S_{spc} = \{\langle a, [s, s_{ac}]\rangle, \langle c, [s_{ac}, e]\rangle, \langle c, [s, s_{bc1}]\rangle, \langle b, [s_{bc1}, s_{bc2}]\rangle, \langle c, [s_{bc2}, e]\rangle\}$ is obtained. Since $S_{spc}' = \{\langle a, [s, s_{ac}]\rangle, \langle c, [s, s_{bc1}]\rangle\}$ with $R_{max} = [s, s_{bc1}]$ having c as the control point, SPC-CO*k*NN first derives $cp'' = b$ and $p'' = p_2$, and then invokes SPC-CO*k*NN (*TRL*, $[s, s_{bc1}]$, $\{p_1, p\}$, $\{a, c\}$, p_2, b) and SPC-CO*k*NN (*TRL*, $[s_{bc1}, e]$, $\{p_1, p_2\}$, $\{a, b\}$, p, c) to further evaluate the impact of b and c on intervals $[s, s_{bc1}]$ and $[s_{bc1}, e]$, respectively. Next, SPC-CO*k*NN (*TRL*, $[s, s_{bc1}]$, $\{p_1, p\}$, $\{a, c\}$, p_2, b) is conducted. Similarly, the Split function is called with SPC(a, b, $[s, s_{bc1}]$) = $\{\langle a, [s, s_{bc1}]\rangle\}$ (notice that the actual split point s_{ab} corresponding to a and b is located outside the interval $[s, s_{bc1}] \subseteq q$ but inside q.), SPC(c, b, $[s, s_{bc1}]$) = $\{\langle c, [s, s_{bc1}]\rangle\}$, and $S_{spc} = \{\langle a, [s, s_{bc1}]\rangle, \langle c, [s, s_{bc1}]\rangle\}$, due to the interval $[s, s_{bc1}] \neq \varnothing$. Here, $S_{spc}' = S_{spc} = \{\langle a, [s, s_{bc1}]\rangle, \langle c, [s, s_{bc1}]\rangle\}$, in which there is no

any interval that has b as its control point. Thus, $R_{min} = [s, s_{bc1}]$ is derived, and the tuple $\langle \{p_1, p\}, \{a, c\}, [s, s_{bc1}] \rangle$ is inserted into *TRL* (without any changing). The algorithm proceeds in the same manner until all the split points along q are identified, after which *TRL* is updated to $\{\langle \{p_1, p\}, \{a, c\}, [s, s_{bc1}] \rangle, \langle \{p_1, p_2\}, \{a, b\}, [s_{bc1}, s_{bc2}] \rangle, \langle \{p_1, p\}, \{a, c\}, [s_{bc2}, s_{ab}] \rangle, \langle \{p_2, p\}, \{b, c\}, [s_{ab}, e] \rangle\}$, as illustrated in Figure 12(a). For ease of understanding, Figure 12(b) shows the executive processes of SPC-CO*k*NN algorithm for the running example.

It is worth mentioning that k has a direct impact on the size of the result list *RL*. In particular, the bigger the k is, the larger the number of intervals/regions included in *RL* is, and the higher the cost incurred by CO*k*NN search algorithm is.

5.2 Trajectory ONN Search

So far we have discussed CONN and CO*k*NN query processing for a *single* query line segment. However, in practice, the users may want to retrieve the ONN of every point on a given *trajectory* that consists of several consecutive line segments. Motivated by this, we introduce *trajectory obstructed nearest neighbor* (TONN) search, which finds the ONN for every point on a specified query trajectory.

An intuitive solution to TONN search, namely *Simple Processing Algorithm* (SP), is to, for each line segment q_i included in a trajectory q (i.e., $\forall\ q_i \subseteq q$), invoke the CONN algorithm to retrieve the ONN 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. This is because, given a query trajectory q that contains λ line segments q_i (i.e., $q = \cup_{1 \le i \le \lambda} q_i$), SP needs to traverse the data R-tree T_p and the obstacle R-tree T_o λ times, resulting in extremely high I/O overhead, especially when λ is large. To address this, we adopt a batch process to evaluate λ line segments simultaneously, such that both T_p and T_o are traversed, respectively, only once. In this way, the I/O cost can be reduced.

In the sequel, we explain how to extend the CONN algorithm to tackle the TONN query by traversing T_p and T_o *only once* no matter how large λ is. The proposed algorithm outperforms SP, as demonstrated in Section 7.3. 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 given query trajectory q is defined as the minimal distance among all the minimal distances (i.e., *mindist*) from E to each line segment $q_i \subseteq q$ ($i \in [1, \lambda]$), i.e., $minimummindist(E, q) = \text{MIN}_{1 \le i \le \lambda}(mindist(E, q_i))$. As depicted in Figure 13, the query trajectory $q = [s, e]$ consists of three consecutive line segments $q_1 = [s, tp_1]$, $q_2 = [tp_1, tp_2]$, and $q_3 = [tp_2, e]$, and the corresponding $minimummindist(p_1, q) = \text{MIN}(mindist(p_1, q_1), mindist(p_1, q_2), mindist(p_1, q_3)) = \text{MIN}(dist(p_1, tp_1), dist(p_1, tp_2), dist(p_1, tp_2)) = dist(p_1, tp_2)$.

Second, as implied by Theorem 4.1 (presented in Section 4.1), the obstacles o that might affect the obstructed distances between a data point p and any point on a specified query line segment $q_i = [q_i.l, q_i.r]$ must have their *mindist* to q bounded by $d = 1/2 \cdot (\text{MIN}(||p, q_i.l||, ||p, q_i.r||) + |q_i| + mindist(p, q_i))$, i.e., $mindist(o, q_i) \le d$. Since we need to find out all the obstacles that may affect the obstructed distances from p to a given query trajectory q consisting of multiple line segments q_i, d should be replaced by $1/2 \cdot (\text{MIN}_{1 \le i \le (\lambda+1)}(||p, tp_i||) + |q| + minimummindist(p, q))$, in which tp_i ($i \in [1, \lambda + 1]$) is an endpoint of a line segment included in q. Take Figure 13 as an example again. All the obstacles o affecting the obstructed distances from p_1 to any point along the query trajectory q must have their *minimummindist* to q, i.e., $minimummindist(o, q)$, bounded by $d = 1/2 \cdot (\text{MIN}(||p_1, s||, ||p_1, tp_1||, ||p_1, tp_2||, ||p_1, e||) + minimummindist(p_1, q) + |q|) = 1/2 \cdot$

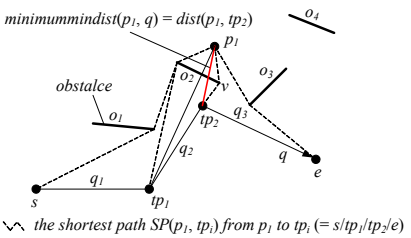


Fig. 13. Distance metric of TONN search

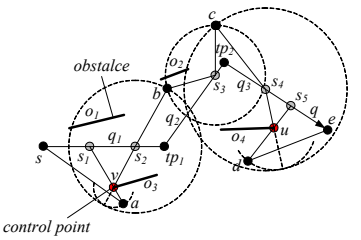


Fig. 14. Example of a TONN query

($||p_1, tp_2|| + |q| + minimummindist(p_1, q)) = 1/2 \cdot (dist(p_1, v) + dist(v, tp_2) + |q| + dist(p_1, tp_2)$).

Third, the lemmas used by the CONN algorithm are still applicable, but the *mindist* metric needs to be replaced by the *minimummindist* metric.

Figure 14 shows an example, where a data set $P = \{a, b, c, d\}$, an obstacle set $O = \{o_1, o_2, o_3, o_4\}$, and a query trajectory $q = [s, e]$ that consists of three consecutive line segments $q_1 = [s, tp_1]$, $q_2 = [tp_1, tp_2]$, and $q_3 = [tp_2, e]$. As illustrated in Figure 14, after processing points b, c, a, d (in this order), the final result of this TONN query is $\{\langle a, a, [s, s_1]\rangle, \langle a, v, [s_1, s_2]\rangle, \langle b, b, [s_2, s_3]\rangle, \langle c, c, [s_3, s_4]\rangle, \langle d, u, [s_4, s_5]\rangle, \langle d, d, [s_5, e]\rangle\}$.

It is worth noting that, the cost of TONN search, compared to CONN retrieval, is higher due to the fact that the number of split points and the number of obstacles retrieved increase with the number of query line segments.

6. APPROXIMATE CONN SEARCH

In general, if both the data and the obstacle sets are huge, the processing of CONN queries incurs extremely high cost since in the worst case it may require evaluating all the data points in P and accessing all the obstacles in O. Motivated by this observation, we, in this section, explore *approximate continuous obstructed nearest neighbor* (ACONN) search (defined in Definition 6.1), which aims to compromise accuracy for efficiency by proposing a method for *fast* but *approximate* CONN retrieval, i.e., the result of ACONN search may incur *false negative* or/and *false positive*.

Definition 6.1 (*Approximate Continuous Obstructed Nearest Neighbor Query*). Given P, O, and q, an *approximate continuous obstructed nearest neighbor* (ACONN) *query* returns the *approximate* result list *ARL* that contains a set of $\langle p_i, R_i\rangle$ ($i \in [1, t]$) tuples where p_i might not necessarily be the ONN to all the points along R_i.

A naive solution to ACONN retrieval is based on sampling. In particular, this method finds the ONN objects for several predefined sample points of a specified query line segment q, and then computes the split points assuming the retrieved ONN objects constitute the final answer set. Nevertheless, due to the natural disadvantage of sampling, its performance highly depends on the number and the positions of sampling points, and the accuracy cannot be guaranteed. In one extreme, we can select one point p from q and assume the ONN to p is the ONN to any point along q. This sampling replaces a CONN query with an ONN query, which incurs much lower cost. However, the accuracy is very low. In the other extreme, we select all the points on q as sampling points, which guarantees the accuracy but results in extremely high processing cost. As a result, it is very difficult to determine an appropriate number and the positions of sampling points

along q, in order to efficiently process ACONN search with high accuracy. Furthermore, it needs to traverse the data R-tree T_p and the obstacle R-tree T_o *multiple times* because it has to conduct ONN retrieval on each sample point, which incurs high I/O overhead.

Actually, our proposed CONN algorithm could be adapted to support ACONN search. As proved in Theorem 4.1 (presented in Section 4.1), only the obstacles with their *mindist* to q bounded by $d = 1/2 \cdot (\text{MIN}(||p, s||, ||p, e||) + |q| + mindist(p, q))$ may affect the obstructed distance between p and any point along a given query line segment $q = [s, e]$. Nonetheless, most of the qualified obstacles that require accessing are inside a much smaller range, as demonstrated in Lemma 6.1.

LEMMA 6.1. *Given a data point p and a query line segment* $q = [s, e]$, *let* $ASR_{p,q}$ *be the range bounded by the shortest path* $SP(p, s)$ *from p to s, the shortest path* $SP(p, e)$ *from p to e, and q. For any point s′ on q, it has high probability that the shortest path* $SP(p, s')$ *from p to s′ only passes the vertexes of the obstacles o that overlap* $ASR_{p,q}$, *i.e.*, $o \cap ASR_{p,q} \neq \emptyset$.

PROOF. Pease refer to Appendix M. □

Based on Lemma 6.1, a *tighter* search range for the obstacles that may affect the obstructed distance between the current data point p evaluated and any point along a specified query line segment $q = [s, e]$ is determined, as stated in Lemma 6.2.

LEMMA 6.2. *Given a data point p and a query line segment* $q = [s, e]$, *let* $ASR_{p,q}$ *be the range bounded by the shortest path* $SP(p, s)$ *from p to s, the shortest path* $SP(p, e)$ *from p to e, and q. Assume* $\delta = \text{MAX}_{\forall v \in V_p}(mindist(v, q))$, *where* V_p *denotes the set of vertexes v (including p, s, and e) on* $SP(p, s)$ *and* $SP(p, e)$. *All the obstacles o with* $mindist(o, q) > \delta$ *do not overlap* $ASR_{p,q}$, *i.e., if* $mindist(o, q) > \delta$, $o \cap ASR_{p,q} = \emptyset$.

PROOF. Pease refer to Appendix N. □

Our proposed *approximated* CONN (ACONN) algorithm for ACONN retrieval, like CONN, evaluates data points according to the ascending order of their *mindist* to q, but (unlike CONN) it retrieves only those obstacles having their *mindist* to q bounded by $\delta = \text{MAX}_{\forall v \in V_p}(mindist(v, q))$ when evaluating a point p, with set V_p containing the set of vertexes v (including p, s, and e) on $SP(p, s)$ and $SP(p, e)$. For example, in Figure 15, $V_p = \{p, v_1, v_2, s, v_3, v_4, e\}$ and $\delta = \text{MAX}_{\forall v \in V_p}(mindist(v, q)) = \text{MAX}(mindist(p, q), mindist(v_1, q), mindist(v_2, q), mindist(s, q), mindist(v_3, q), mindist(v_4, q), mindist(e, q)) = mindist(p, q) = dist(p, s_1)$. According to Lemma 6.2, the shortest distance from p to any point along q is most likely affected only by the obstacles with their *mindist* to q bounded by $mindist(p, q) = dist(p, s_1)$.

Compared with CONN, ACONN saves the retrieval of those obstacles o with $mindist(o, q)$ larger than δ but bounded by $d = 1/2 \cdot (\text{MIN}(||p, s||, ||p, e||) + |q| + mindist(p, q))$. Therefore, ACONN may miss some obstacles (e.g., obstacle o'' in Figure 15) that do affect the obstructed distances between the evaluated data point p and some points on q. However, as demonstrated in Section 7.4, the number of missed obstacles is relatively *small*, compared with the performance improvement. In addition, ACONN can be naturally extended to answer approximate CO*k*NN (ACO*k*NN) search. The details are skipped to save space.

7. EXPERIMENTAL EVALUATION

In this section, we verify the performance of our proposed algorithms for CONN query and its variants via extensive experimental evaluation. All the algorithms were

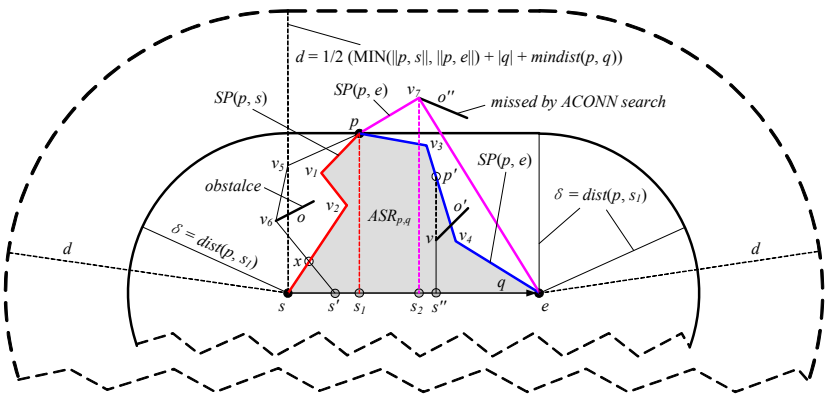


Fig. 15. Obstacle search ranges for ACONN search

implemented in C++, and the experiments were conducted on an Intel Core 2 Duo 3.2 GHz PC with 3GB RAM, running Microsoft Windows XP Professional Edition. The detailed experimental setup is presented in Section 7.1. Section 7.2 studies the efficiency and effectiveness of CONN algorithm in supporting COkNN ($k \geq 1$) search. Sections 7.3 and 7.4 evaluate TOkNN and ACOkNN queries, respectively.

7.1 Experimental Settings

Our experiments use both real and synthetic datasets, with the search/data space fixed at a [0, 10000] × [0, 10000] square. Two real datasets are deployed, namely, *CA* and *LA*[16]. Specifically, *CA* contains two-dimensional (2D) points, representing 60,344 locations in California; and *LA* includes 2D rectangles, representing 131,461 MBRs of streets in Los Angeles. All 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×|*LA*| to 10×|*LA*|. The coordinate of each point in *Uniform* datasets is created uniformly along each dimension, and that of each point in *Zipf* datasets is generated according to a zipf distribution with a skew coefficient[17] $\alpha = 0.8$. We assume a point's coordinates on both dimensions are mutually independent.

Since CONN search and its variants involve a data set P and an obstacle set O, we deploy three different combinations of the datasets, namely ***CL***, ***UL***, and ***ZL***, representing (P, O) = (*CA*, *LA*), (*Uniform*, *LA*), and (*Zipf*, *LA*), respectively. *CL* utilizes real datasets with $|P| < |O|$. On the other hand, *UL* and *ZL* employ synthetic datasets, and thus we can adjust the relative size of P and O to simulate different cases. 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 [Beckmann et al. 1990], with a page size of 4K bytes. Table II lists all the parameters that are considered in our 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 100 queries for each experiment, and the average performance is reported.

[16] *CA* and *LA* datasets are available in the R-tree portal (http://www.rtreeportal.org).
[17] When the skew coefficient equals 1, all numbers generated by the Zipf distribution are equivalent. When the coefficient equals 0, the Zipf distribution degenerates to uniformity.

Table II. Parameter ranges and default values

Parameter	Range
query length $\|q\|$ (% of data space side)	1.5, 3, **4.5**, 6, 7.5
k	1, 3, **5**, 7, 9
$\|P\|/\|O\|$	0.1, 0.2, 0.5, **1**, 2, 5, 10
buffer size (% of the tree size)	**0**, 1, 2, 4, 8, 16, 32
number of trajectory segments λ	2, 3, **4**, 5, 6

The main performance metrics in our experimental evaluation include I/O cost (i.e., the number of node/page accesses), CPU time, total query cost (i.e., the sum of the I/O time and CPU time, where the I/O time is calculated by charging 10ms for each page access [Tao et al. 2007]), the number of data points evaluated (NPE) during search, the number of obstacles encountered (NOE) during search, and visibility graph size $|VG|$ (i.e., the number of vertexes contained in the local visibility graph VG). Unless specifically stated, the size of LRU buffer is 0 in the experiments, i.e., the I/O cost is determined by the number of nodes/pages accessed. Given a query length $|q|$, 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 $|q|$. In addition, we assume each query line segment does not cross any obstacle. The line segments included into a given query trajectory for TOkNN retrieval are created in the same manner. However, we fix the trajectory length to 4.5% of the data 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 Evaluation of COkNN Search

The first set of experiments aims at evaluating the performance of CONN algorithm in answering COkNN queries. We study the influence of various parameters, including (i) query length $|q|$, (ii) the number of ONNs 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 4, the data set P and the obstacle set O can be indexed by two *separate* R-trees or a *single* R-tree. Consequently, the performance of COkNN search under both settings, denoted as COkNN-2T (2T for short) and COkNN-1T (1T for short) respectively, is evaluated as well.

First, we investigate the effect of $|q|$ on the efficiency of the algorithms using *CL* dataset combination. Figure 16 shows the performance of the COkNN algorithm in terms of I/O cost, total query cost (in seconds), number of items evaluated (including NPE and NOE), and visibility graph size, respectively, as a function of $|q|$, fixing k to 5. Here, the I/O cost is broken to data R-tree node accesses and obstacle R-tree node accesses, respectively; and the overall query cost is also broken into two components, corresponding to the I/O cost and the CPU cost, respectively. Furthermore, we explicitly point out the percentage of obstacle R-tree node accesses in the total node accesses (i.e., the summation of data R-tree and obstacle R-tree node accesses during the search) and the percentage of I/O cost in the overall query cost, denoted by the percentage number on top of each bar in Figure 16(a) and Figure 16(b), respectively.

It is observed that the cost of COkNN retrieval grows with $|q|$. The reason behind is that, as the length of query line segment becomes longer, the number of data points processed, the number of obstacles encountered (i.e., the ones overlapping the obstacle search range), and the number of split points generated along the specified query line

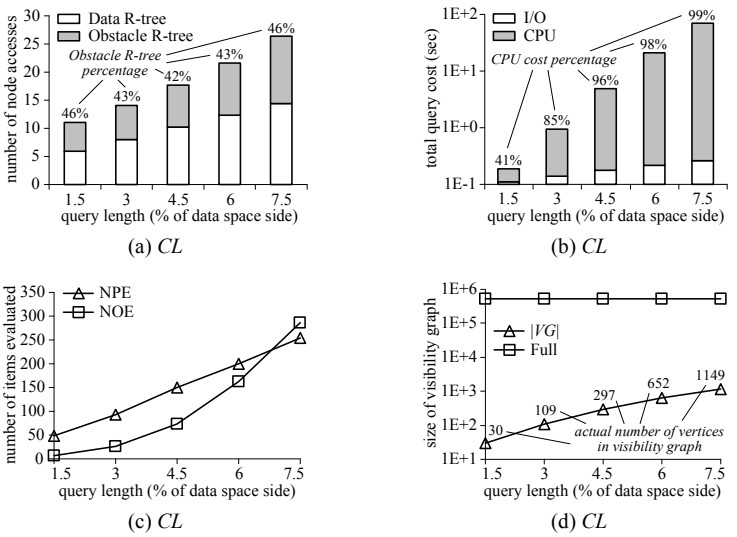


Fig. 16. CO*k*NN search performance vs. query length $|q|$ ($k = 5$)

segment increase, resulting in more distance calculation, more control point list computation, and more result list update. Figures 16(a), 16(b), and 16(c) further confirm this explanation, showing that the number of node accesses, CPU cost percentage, NPE, and NOE ascend with the growth of $|q|$. Figure 16(d) illustrates the size of visibility graph (i.e., $|VG|$) with respect to $|q|$. Since all the obstacles are in rectangular shapes (as specified in the footnote 1 of Section 1), there are $4 \times |O| = 525{,}844$ vertexes in the global visibility graph, denoted by FULL, when we take *LA* as the obstacle set *O*. Note that, although $|VG|$ increases with $|q|$, its size is *much smaller* than the size of FULL, as also demonstrated in the subsequent experiments. This confirms the effectiveness of our proposed IOR algorithm in reducing the number of obstacle traversals because IOR only incrementally retrieves all the qualified obstacles that may affect the final query result.

Next, we explore the impact of *k* on the efficiency of the algorithms. Towards this, we employ *CL* again, set $|q|$ to 4.5% of the data space side, and vary *k* between 1 and 9. Figure 17 depicts the performance of the CO*k*NN algorithm for various values of *k*. As expected, all costs, including the number of node accesses, total query time, NPE, NOE, and $|VG|$, grow with *k*. This is because a higher value of *k* implies a larger range to be searched (for both data points and obstacles), incurring more distance computation. Moreover, as *k* increases, the number of answer points in the final result list *RL* grows as well, which leads to more frequent update operations and more expensive maintenance cost for the result list *RL*. In addition, notice that the ascending trend of Figure 17 is not as obvious as that observed from Figure 16, which indicates that the change of *k* has a smaller impact on the search performance than the change of $|q|$.

Then, to study the influence of $|P|/|O|$, we utilize *UL* and *ZL* dataset combinations, where the ratio of the cardinality of the data set *P* to that of the obstacle set *O*, i.e., $|P|/|O|$, varies from 0.1 to 10. Figure 18 plots the performance of the CO*k*NN algorithm as a function of the ratio $|P|/|O|$, with $|q| = 4.5\%$ and $k = 5$. A crucial observation is that the total query cost of CO*k*NN search first drops and then increases as $|P|/|O|$ varies. Specifically, CO*k*NN retrieval decreases its query cost when $|P|/|O|$ grows from 0.1 to 0.5

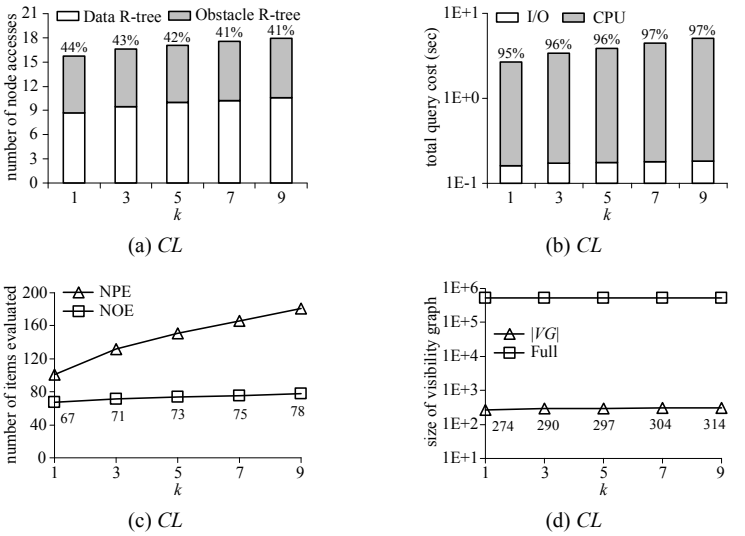


Fig. 17. CO*k*NN search performance vs. k ($|q|$ = 4.5%)

(see Figures 18(b) and 18(f)). This is because, as the density of data set P increases, the search space of CO*k*NN search becomes smaller. Correspondingly, the number of obstacles that might impact the obstructed distances from every data point to any point along a given query line segment q is decreased, i.e., the IOR algorithm retrieves less qualified obstacles, which is indicated by NOE in Figures 18(c) and 18(g). Nevertheless, as $|P|/|O|$ continues growing from 1 to 10 (see Figures 18(b) and 18(f)), the query cost of CO*k*NN retrieval increases gradually. The reason behind is that, the interval/region on q dominated by each data point becomes smaller, and the result list contains more answer points. In other words, more data points need to be evaluated, as implied by NPE in Figures 18(c) and 18(g), with higher split point computation overhead and higher result list update cost. Notice that, when P and O share similar cardinalities (e.g., $|P|/|O|$ = 0.5 or 1), CO*k*NN search takes the smallest query time.

We also observe that from Figures 18(a) and 18(e), with the growth of $|P|/|O|$, the node/page accesses of the data R-tree increase fast, whereas those of the obstacle R-tree decline gradually. This is because, as $|P|/|O|$ ascends, the range around the query line segment where answer points are found decreases. Figures 18(c) and 18(g) further confirm this observation, showing that NPE grows fast, but NOE drops gradually. In addition, as illustrated in Figures 18(d) and 18(h), the visibility graph size $|VG|$ drops as $|P|/|O|$ increases, due to the shrinking of the CO*k*NN search space.

As mentioned in Section 7.1, all the previous experiments are conducted without any buffer, i.e., the size of LRU buffer is 0. In this set of experiments, we examine the effect of buffer size on the CO*k*NN search performance, by changing the buffer size from 0% to 32% of each R-tree size. To obtain stable statistics, we use the first 50 queries to warm up the buffer, and only measure the average cost of the last 50 queries. Figure 19 depicts the results for the *CL* dataset combination, with $|q|$ = 4.5% and k = 5. It is observed that a non-zero buffer can improve the I/O performance only, but not others. In particular, as the buffer size increases, the I/O cost decreases gradually (see Figure 19(a)), whereas the other costs (e.g., CPU cost, etc.) remain almost the same.

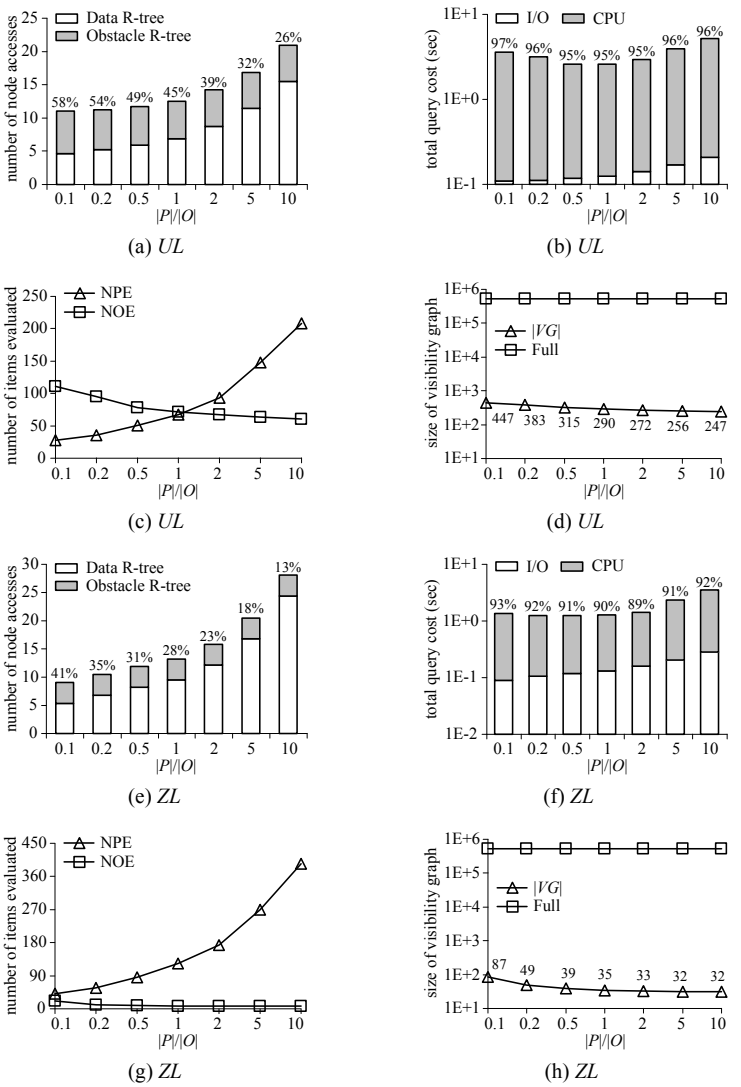


Fig. 18. CO*k*NN search performance vs. $|P|/|O|$ ($|q|$ = 4.5%, k = 5)

In all the above experiments, we assume that the data set P and the obstacle set O are indexed by two separate R-trees. However, as discussed in Section 4.5, our proposed CO*k*NN search algorithm is very flexible, and it can easily support the case where both P and O are indexed by a single R-tree. The last set of experiments in this subsection compares the performance of CO*k*NN retrieval when P and O are indexed by two different R-trees (i.e., CO*k*NN-2T (2T for short)) against that under the scenario where both P and O are indexed by one unified R-tree (i.e., CO*k*NN-1T (1T for short)), and the experimental results are plotted in Figure 20. It shows that 1T is more efficient than 2T in most of the cases, although they share the similar performance trend. This is because, when both data points and obstacles are indexed by one R-tree, only one traversal of the

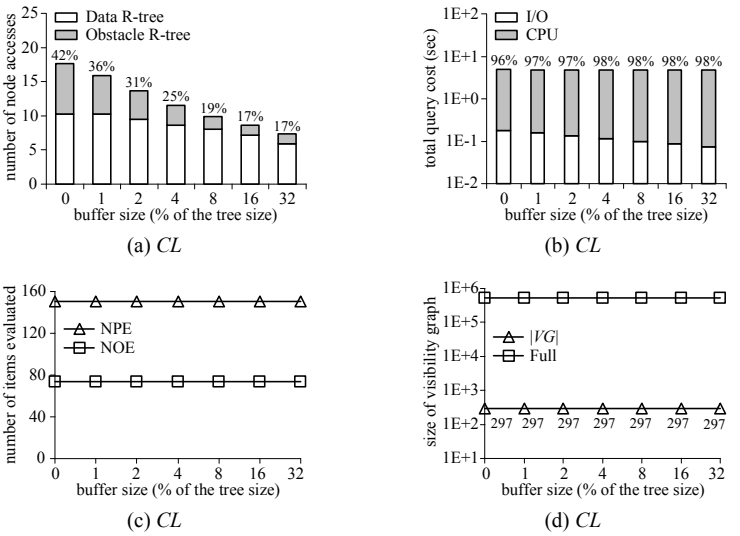


Fig. 19. CO*k*NN search performance vs. buffer size (|*q*| = 4.5%, *k* = 5)

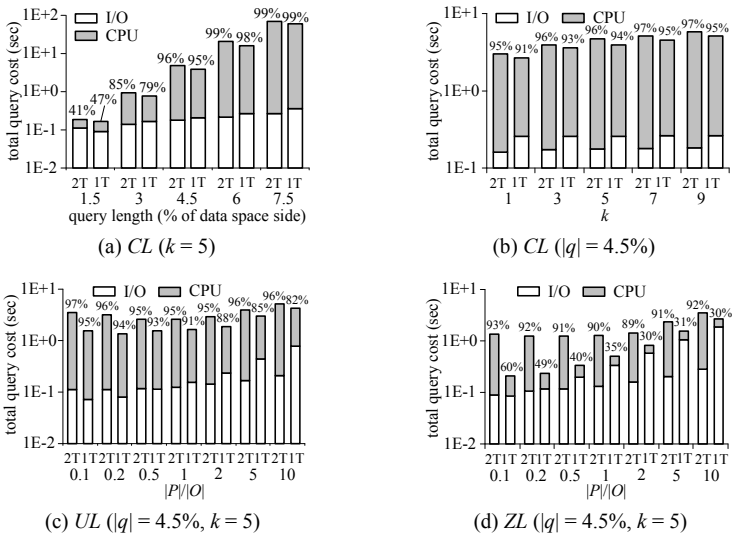


Fig. 20. CO*k*NN search performance on two R-trees vs. that on one R-tree

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 both *P* and *O* is one potential approach to further boost up the search performance. It is observed that 1T incurs a high I/O cost than 2T in most of the cases. The reason behind is that, when both data points and obstacles are indexed by a single R-tree, more data points are likely to be visited/evaluated during the search (although they are unqualified ones), leading to more I/O overhead. We also observe that the performance gap between 2T and 1T is more

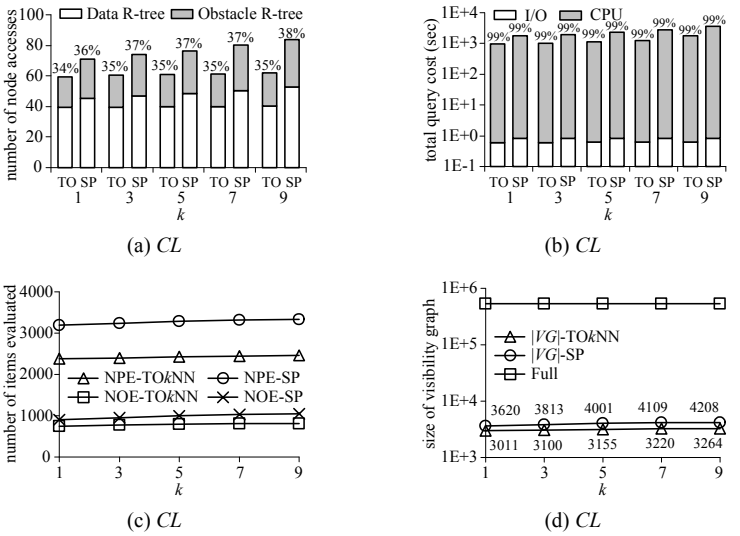


Fig. 21. TO*k*NN search performance vs. k ($\lambda = 4$)

obvious under *ZL* than that for *UL*, due to the different dataset distribution. In addition, the performance difference between 2T and 1T is more significant when $|P|/|O|$ is lower. This is because, when the density of data set P is smaller, the search space of CO*k*NN retrieval becomes larger, resulting in higher traversal cost of R-trees.

7.3 Evaluation of TO*k*NN Search

The second set of experiments evaluates the performance of our proposed algorithm for TO*k*NN search (called TO*k*NN). For comparison, we implement the SP algorithm (presented in Section 5.2) as a baseline approach. We study the influence of two factors: k and the number of trajectory segments λ. The trajectory length is set to 4.5% of the data space side, and it consists of λ consecutive line segments with equivalent length.

First, we investigate the effect of k on the efficiency of the algorithms. Figure 21 shows the performance of the algorithms as a function of k, fixing λ to 4 (which is the median value used in Figure 22). Clearly, TO*k*NN outperforms SP, especially for the I/O cost, NPE, NOE, and $|VG|$. The reason behind is that, as mentioned in Section 5.2, TO*k*NN answers TO*k*NN retrieval by traversing both the data set P and the obstacle set O only once. Notice that, in terms of overall query cost, although TO*k*NN is better than SP, they are similar, as illustrated in Figure 21(b). This is because the calculation of *minimummindist* metric (proposed in Section 5.2) requested by TO*k*NN is CPU time-consuming. In addition, the cost of TO*k*NN search increases with k, since a higher value of k incurs a larger search space, more distance computation, and more result list maintenance cost. As SP always performs worse than TO*k*NN, it is omitted from the remaining experiments in this subsection.

Then, we explore the impact of λ on the efficiency of TO*k*NN algorithm. Figure 22 depicts the results as a function of λ using the *CL* dataset combination, by fixing k to 5 and varying λ from 2 to 6. As expected, the cost of the algorithm increases with the growth of λ. The reason behind is that, a larger λ suffers from more distance calculation,

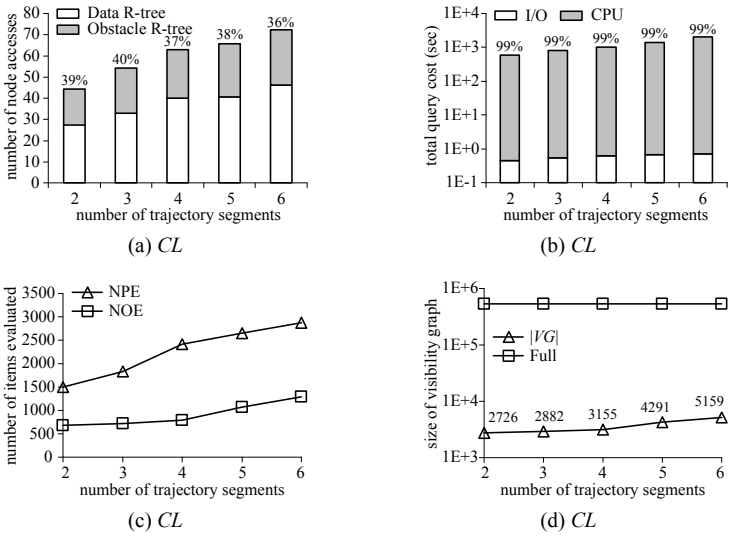


Fig. 22. TO*k*NN search performance vs. the number of trajectory segments λ (k = 5)

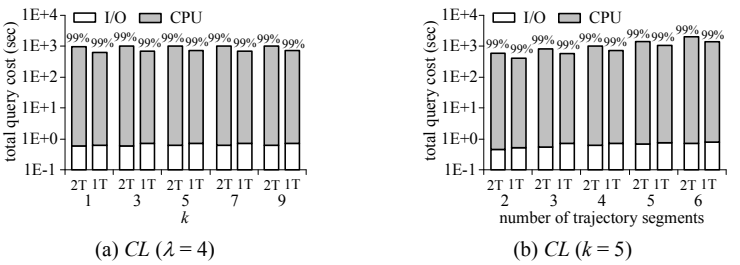


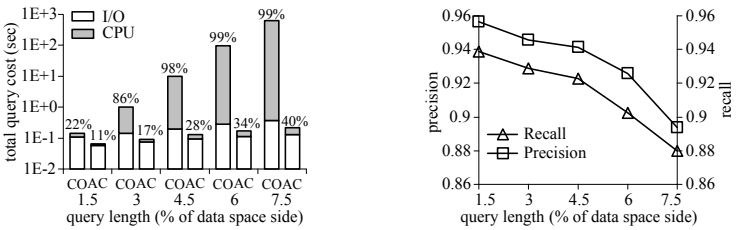
Fig. 23. TO*k*NN search performance on two R-trees vs. that on one R-tree

more control point list computation, and more frequent result list update.

Also, we compare the performance of TO*k*NN when the data set P and the obstacle set O are indexed by two separate R-trees, denoted as TO*k*NN-2T (2T for short), against that under the case where both P and O are indexed by a single R-tree, denoted as TO*k*NN-1T (1T for short). The results are plotted in Figure 23. It is observed that TO*k*NN-2T and TO*k*NN-1T share the same performance trend, whereas the latter performs better. As mentioned earlier, the advantage of TO*k*NN-1T can be explained by the fact that the data points and obstacles located close to each other are very likely to be stored in the same page, and hence, the access to the data points and that to the obstacles might share the node access when both P and O are indexed by a unified R-tree.

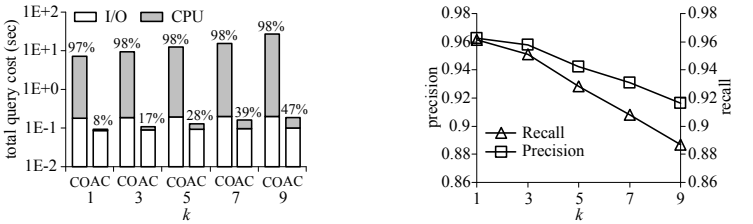
7.4 Evaluation of ACO*k*NN Search

The last set of experiments aims to evaluate the efficiency of ACO*k*NN algorithm in answering ACO*k*NN queries. For the approximate algorithm, in addition to the search efficiency, we need to evaluate the quality of approximation. Towards this, we employ two metrics, i.e., *precision* and *recall*. Let ARL_o be the set of answer objects contained in



(a) *CL* (b) *CL*

Fig. 24. ACO*k*NN search performance vs. query length $|q|$ (k = 5)



(a) *CL* (b) *CL*

Fig. 25. ACO*k*NN search performance vs. k ($|q|$ = 4.5%)

the final result list of the approximate search (i.e., ACO*k*NN (AC for short)) and RL_o be the set of answer objects included in the final result list of the exact search (i.e., CO*k*NN (CO for short)). The precision and recall are defined as follows: *precision* = $|ARL_o \cap RL_o| / |ARL_o|$ and *recall* = $|ARL_o \cap RL_o| / |RL_o|$. Note that, given the fact that the result of CO*k*NN search is a set of $\langle p_i, R_i \rangle$ tuples, we use $\langle p_i, R_i \rangle$ instead of a single p_i to measure $|ARL_o \cap RL_o|$. For example, let $ARL_o = \{\langle p_1, R_1 \rangle, \langle p_2, R_2 \rangle, \langle p_3, R_3 \rangle, \langle p_4, R_4 \rangle\}$ and $RL_o = \{\langle p_1, R_1' \rangle, \langle p_2, R_2' \rangle, \langle p_3, R_3 \rangle, \langle p_5, R_4 \rangle\}$ ($R_1 \neq R_1'$ and $R_2 \neq R_2'$), then $|ARL_o \cap RL_o| = |\{\langle p_3, R_3 \rangle\}| = 1$. In addition, it is worth mentioning that, we do not consider the sampling based approach for ACO*k*NN search because, as mentioned in Section 6, it has two deficiencies: (i) it is difficult to identify appropriate number and positions of sampling points in order to obtain high accuracy guarantee; and (ii) it may repetitively access some node entries in R-trees.

In the first experiment, we fix k to 5 and vary $|q|$ from 1.5% to 7.5% of the data space side. Figure 24(a) illustrates the total query cost with respect to $|q|$. Evidently, ACO*k*NN outperforms CO*k*NN and the performance difference becomes more significant as $|q|$ increases. Figure 24(b) unveils the precision and recall incurred by ACO*k*NN as a function of $|q|$. For a short query length (e.g., $|q|$ = 1.5%), the precision and recall are high since the number of qualifying obstacles missed is limited. As a result, the difference between actual obstacle search range and approximate obstacle search range is insignificant. On the other hand, the number of missing qualified obstacles grows dramatically as $|q|$ increases.

Figure 25 shows the total query cost, precision, and recall under different k, by fixing $|q|$ to 4.5% of the data space side. Similar to Figure 24, ACO*k*NN outperforms CO*k*NN significantly in terms of performance, with reasonable precision and recall. Even when k increases, the precision and recall of ACO*k*NN are still acceptable.

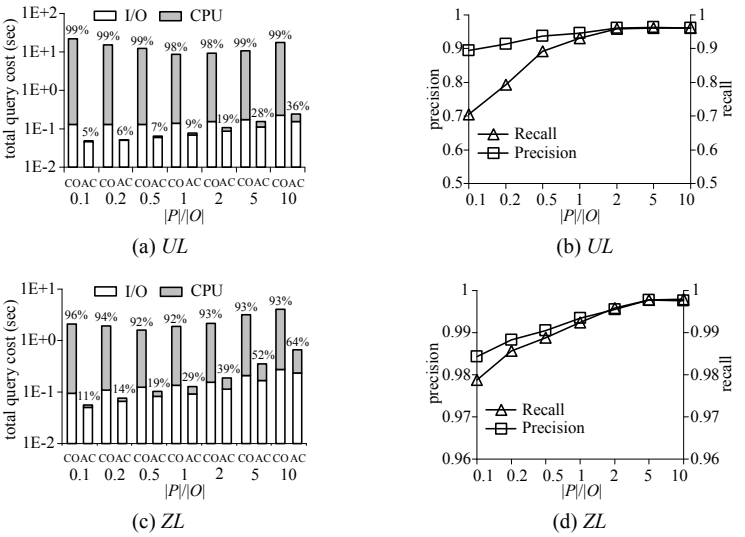


Fig. 26. ACO*k*NN search performance vs. $|P|/|O|$ ($|q|$ = 4.5%, k = 5)

Finally, we demonstrate the effect of $|P|/|O|$ on the performance of the algorithms, using synthetic dataset combinations *UL* and *ZL*. The results are plotted in Figure 26. Again, ACO*k*NN is consistently better than CO*k*NN for all the experimental instances. As shown in Figures 26(b) and 26(d), the precision and recall increase as $|P|/|O|$ grows. This is because, when $|P|$ is smaller than $|O|$, the obstacle set O has a larger density than P, and hence more obstacles actually fall outside the approximate obstacle search range but within the real obstacle search range, i.e., the number of qualified obstacles ignored by ACO*k*NN but not CO*k*NN is larger. Moreover, we observe that the total query cost for CO*k*NN search and that for ACO*k*NN retrieval have different performance trend (see Figures 26(a) and 26(c)), due to the different obstacle search range.

8. CONCLUSIONS

In this article, we identify and solve a novel type of CNN queries, namely CONN search, which considers the impact of obstacles on the distance between objects. CONN retrieval is not only interesting and challenging from a research point of view, but also useful in many applications such as location-based commerce, geographic information systems, and complex spatial data analysis under obstacle constraints. We carry out a systematic study of CONN search. First, we provide a formal definition of the problem. Second, we propose efficient algorithms for exact CONN query processing. Next, we extend our techniques to tackle variations of CONN queries, including CO*k*NN search and trajectory ONN search. Then, we discuss approximate CONN retrieval. Finally, we conduct extensive experiments to evaluate the efficiency and effectiveness of the proposed algorithms using both real and synthetic datasets.

In the future, we intend to explore the application of the related techniques to other forms of spatial queries (e.g., reverse nearest neighbor search [Korn and Muthukrishnan 2000; Tao et al. 2007], etc.) in the presence of obstacles. Another promising direction for future work concerns the extension of our proposed methodology to alternative versions

of the problem. One such example refers to CONN queries for moving objects or/and moving obstacles. Finally, it would be particularly interesting to develop analytical models for estimating the execution cost of CONN search algorithms, because such models will not only facilitate query optimization, but may also reveal new problem characteristics that could lead to even better algorithms.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: http://www.acm.rog/pubs/citations/journals/tods/20XX-X-X/p1-URLend.

REFERENCES

BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 322–331.

BERG, M. DE, KREVELD, M. VAN, OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry: Algorithms and Applications*, *Second Edition*. Springer-Verlag.

CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 379–390.

CHEUNG, K. L. AND FU, A. W-C. 1998. Enhanced nearest neighbour search on the R-tree. *SIGMOD Rec. 27*, 3, 16–21.

CHO, H.-J. AND CHUNG, C.-W. 2005. An efficient and scalable approach to CNN queries in a road network. In *Proceedings of the International Conference on Very large Data Bases* (*VLDB*). 865–876.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

ESTIVILL-CASTRO, V. AND LEE, I. 2000. Autoclust+: Automatic clustering of point-data sets in the presence of obstacles. In *Proceedings of the International Workshop on Temporal, Spatial, and Spatio-Temporal Data Mining* (*TSDM*). 133–146.

FENG, J. AND WATANABE, T. 2002. A fast method for continuous nearest target objects query on road network. In *Proceedings of the International Conference on Virtual Systems and Multimedia* (*VSMM*). 182–191.

GAO, Y. AND ZHENG, B. 2009. Continuous obstructed nearest neighbor queries in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 577–590.

GAO, Y., ZHENG, B., CHEN, G., LEE, W.-C., LEE, KEN C. K., AND LI, Q. 2009a. Visible reverse *k*-nearest neighbor query processing in spatial databases. *IEEE Trans. Knowl. Data Engin. 21*, 9, 1314–1327.

GAO, Y., ZHENG, B., LEE, W.-C., AND CHEN, G. 2009b. Continuous visible nearest neighbor queries. In *Proceedings of the International Conference on Extending Database Technology* (*EDBT*). 144–155.

GHOSH, S. K. AND MOUNT, D. M. 1987. An output sensitive algorithm for computing visibility graphs. In *Proceedings of the Annual Symposium on Foundations of Computer Science* (*FOCS*). 11–19.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 47–57.

HENRICH, A. 1994. A distance-scan algorithm for spatial access structures. In *Proceedings of the ACM Workshop on Advances in Geographic Information Systems* (*GIS*). 136–143.

HERSHBERGER, J. AND SURI, S. 1999. An optimal algorithm for euclidean shortest paths in the plane. *SIAM J. Comput. 28*, 6, 2215–2256.

HJALTASON, G. R. AND SAMET, H. 1999. Distance browsing in spatial databases. *ACM Trans. Datab. Syst. 24*, 2, 265–318.

HUANG, Y.-K., LIAO, S.-J., AND LEE, C. 2009. Evaluating continuous *k*-nearest neighbor query on moving objects with uncertainty. *Inf. Syst. 34*, 4-5, 415–437.

IWERKS, G. S., SAMET, H., AND SMITH, K. 2003. Continuous *k*-nearest neighbor queries for continuously moving points with updates. In *Proceedings of the International Conference on Very large Data Bases* (*VLDB*). 512–523.

KOLAHDOUZAN, M. R. AND SHAHABI, C. 2005. Alternative solutions for continuous *k* nearest neighbor queries in spatial network databases. *GeoInformatica 9*, 4, 321–341.

KORN, F. AND MUTHUKRISHNAN, S. 2000. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 201–212.

LI, C., GU, Y., LI, F., AND CHEN, M. 2010. Moving *k*-nearest neighbor query over obstructed regions. In *Proceedings of the Asia-Pacific Web Conference on Advances in Web Technologies and Applications* (*APWeb*). 29–35.

LI, Y., YANG, J., AND HAN, J. 2004. Continuous *k*-nearest neighbor search for moving objects. In *Proceedings of the International Conference on Scientific and Statistical Database Management* (*SSDBM*). 123–126.

LIU, F., HUA, K. A., AND DO, T. T. 2007. A P2P technique for continuous *k*-nearest-neighbor query in road networks. In *Proceedings of the International Conference on Database and Expert Systems Applications* (*DEXA*), 264–276.

MOURATIDIS, K., HADJIELEFTHERIOU, M., AND PAPADIAS, D. 2005a. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 634–645.

MOURATIDIS, K. AND PAPADIAS, D. 2007. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Engin. 19*, 6, 789–803.

MOURATIDIS, K., PAPADIAS, D., BAKIRAS, S., AND TAO, Y. 2005b. A threshold-based algorithm for continuous monitoring of *k* nearest neighbors. *IEEE Trans. Knowl. Data Engin. 17*, 11, 1451–1464.

MOURATIDIS, K., YIU, M., PAPADIAS, D., AND MAMOULIS, N. 2006. Continuous nearest neighbor monitoring in road networks. In *Proceedings of the International Conference on Very large Data Bases* (*VLDB*). 43–54.

NUTANONG, S., TANIN, E., AND ZHANG, R. 2007. Visible nearest neighbor queries. In *Proceedings of the International Conference on Database Systems for Advanced Applications* (*DASFAA*). 876–883.

NUTANONG, S., ZHANG, R., TANIN, E., AND KULIK, L. 2008. The V*-Diagram: A query dependent approach to moving *k*NN queries. *PVLDB 1*, 1, 1095–1106.

NUTANONG, S., ZHANG, R., TANIN, E., AND KULIK, L. 2010. Analysis and evaluation of V*-*k*NN: An efficient algorithm for moving *k*NN queries. *VLDB J. 19*, 3, 307–332.

OKABE, A., BOOTS, B., SUGIHARA, K., AND CHIU, S. N. 2000. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, *Second Edition*. Wiley.

PARK, S. H., LEE, J.-H., AND KIM, D.-H. 2007. Spatial clustering based on moving distance in the presence of obstacles. In *Proceedings of the International Conference on Database Systems for Advanced Applications* (*DASFAA*). 1024–1027.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 71–79.

SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the International Conference on Very large Data Bases* (*VLDB*). 507–518.

SHARIR, M. AND SCHORR, A. 1986. On shortest paths in polyhedral spaces. *SIAM J. Comput. 15*, 1, 193–215.

SISTLA, P., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. 1997. Modeling and querying moving objects. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). 422–432.

SONG, Z. AND ROUSSOPOULOS, N. 2001. *K*-nearest neighbor search for moving query point. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases* (*SSTD*). 79–96.

TAO, Y. AND PAPADIAS, D. 2002. Time parameterized queries in spatio-temporal databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 334–345.

TAO, Y., PAPADIAS, D., LIAN, X., AND XIAO, X. 2007. Multidimensional reverse *k*NN search. *VLDB J. 16*, 3, 293–316.

TAO, Y., PAPADIAS, D., AND SHEN, Q. 2002. Continuous nearest neighbor search. In *Proceedings of the International Conference on Very large Data Bases* (*VLDB*). 287–298.

TERRY, D., GOLDBERG, D., NICHOLS, D., AND OKI, B. 1992. Continuous queries over append-only databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 321–330.

TUNG, A. K. H., HOU, J., AND HAN, J. 2001a. Spatial clustering in the presence of obstacles. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). 359–367.

TUNG, A. K. H., NG, R. T., LAKSHMANAN, L. V. S., AND HAN, J. 2001b. Constraint-based clustering in large databases. In *Proceedings of the International Conference on Database Theory* (*ICDT*). 405–419.

WANG, X. AND HAMILTON, H. J. 2005. Clustering spatial data in the presence of obstacles. *Int. J. Artif. Intell. Tools 14*, 1-2, 177–198.

WANG, X., ROSTOKER, C., AND HAMILTON, H. J. 2004. Density-based spatial clustering in the presence of obstacles and facilitators. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery* (*PKDD*). 446–458.

WU, W., GUO, W., AND TAN, K. L. 2007. Distributed processing of moving *k*-nearest-neighbor query on moving objects. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). 1116–1125.

XIA, C., HSU, D., AND TUNG, A. K. H. 2004. A fast filter for obstructed nearest neighbor queries. In *Proceedings of the British National Conference on Databases* (*BNCOD*). 203–215.

XIONG, X., MOKBEL, M. F., AND AREF, W. G. 2005. SEA-CNN: Scalable processing of continuous *k*-nearest neighbor queries in spatio-temporal databases. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). 643–654.

XU, H., LI, Z., LU, Y., DENG, K., AND ZHOU, X. 2010. Group visible nearest neighbor queries in spatial databases. In *Proceedings of the International Conference on Web-Age Information Management* (*WAIM*). 333–344.

YU, X., PU, K., AND KOUDAS, N. 2005. Monitoring *k*-nearest neighbor queries over moving objects. In *Proceedings of the IEEE International Conference on Data Engineering* (*ICDE*). 631–642.

ZAIANE, O. R. AND LEE, C.-H. 2002. Clustering spatial data in the presence of obstacles: A density-based approach. In *Proceedings of the International Database Engineering and Applications Symposium* (*IDEAS*). 214–223.

ZHANG, J., PAPADIAS, D., MOURATIDIS, K., AND ZHU, M. 2004. Spatial queries in the presence of obstacles. In *Proceedings of the International Conference on Extending Database Technology* (*EDBT*). 366–384.

ZHANG, J., ZHU, M., PAPADIAS, D., TAO, Y., AND LEE, D. L. 2003. Location-based spatial queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (*SIGMOD*). 443–454.

Received February 2010; revised September 2010; accepted November 2010

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Continuous Nearest Neighbor Search in the Presence of Obstacles

YUNJUN GAO
Zhejiang University
BAIHUA ZHENG
Singapore Management University
GANG CHEN and CHUN CHEN
Zhejiang University
and
QING LI
City University of Hong Kong

ACM Transaction on Database Systems, Vol. XX, No. X, Article XX, Publication date: XXXXXXXX 20XX.

A. PROOF OF THEOREM 3.1

PROOF. Consider the illustrative example of Figure 7(a), in which points m and n are the projections of u and v on the line segment q respectively, point y is the intersection between q and the extended line of segment $[u, v]$, and point z is the intersection between the perpendicular bisector $\perp(u, v)$ of $[u, v]$ and q. We further assume that point n is the origin of the XY coordinate system as depicted in Figure 7(a). Let $dist(n, m) = a$ (> 0), $dist(v, n) = b$, $dist(u, m) = c$, and suppose $b < c$ (i.e., v is closer to q than u). As we want to find point(s) s' on q such that $||p, v|| + dist(v, s') = ||p', u|| + dist(u, s')$, we need to find point(s) s' along q that satisfy $dist(u, s') - dist(v, s') = ||p, v|| - ||p', u|| = d$. Suppose point s' on q has coordinate $(x, 0)$, we need to solve the following quadratic polynomial:

$$d = dist(u,s') - dist(v,s') = \sqrt{(a-x)^2 + c^2} - \sqrt{x^2 + b^2} \quad (1)$$

Let $A = 4a^2 - 4d^2$, $B = -4aT$, and $C = T^2 - 4b^2d^2$ with $T = a^2 + c^2 - b^2 - d^2$, the roots of Equation (1) can be derived as follows: (i) if $A = 0$, then $x = -C/B$; otherwise (ii) $x = (-B \pm \sqrt{B^2 - 4AC})/(2A)$. Hence, there are *at most two* points along q such that $||p, v|| + dist(v, s') = ||p', u|| + dist(u, s')$. The proof can be easily adjusted for other cases, including (i) $a = 0$, i.e., $[u, v]$ is vertical to q; (ii) $b = c$, i.e., $[u, v]$ is parallel to q, and (iii) $b > c$, i.e., u is closer to q than v. □

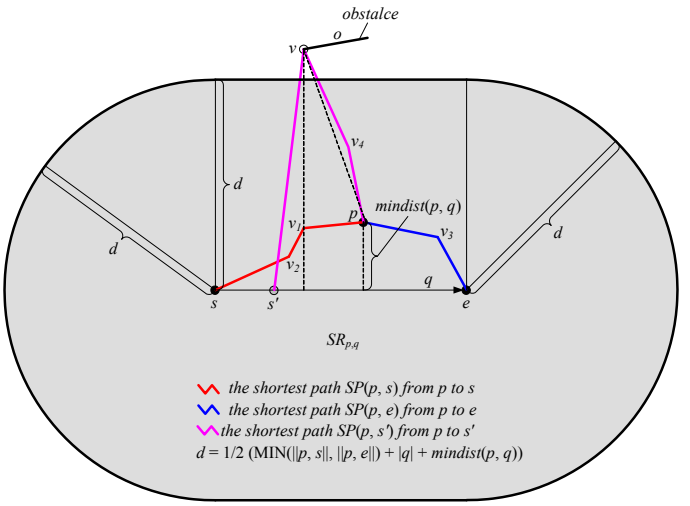


Fig. 27. The obstacle search range

B. PROOF OF LEMMA 3.1

PROOF. Without loss of generality, we assume that there is at least one point s' along the line segment q such that $||p', s'|| < ||p, s'||$ (i.e., p' is nearer to s' than p). As points v and u are the control points of p and p' over q respectively, $||p, s'|| = ||p, v|| + dist(v, s')$ and $||p', s'|| = ||p', u|| + dist(u, s')$. $||p', s'|| < ||p, s'||$ indicates that $||p', u|| + dist(u, s') < ||p, v|| + dist(v, s')$, i.e., $dist(u, s') - dist(v, s') < ||p, v|| - ||p', u|| = d$. On the other hand, based on the conditions (i) and (ii) presented in Lemma 3.1, we have $dist(u, s) - dist(v, s) > ||p, v|| - ||p', u|| = d$ and $dist(u, e) - dist(v, e) > ||p, v|| - ||p', u|| = d$. Let $Y(x) = dist(u, x) - dist(v, x)$ with $x \in [s, e]$. Since x is a certain point along $q = [s, e]$, the value of $Y(x)$ first *drops* and then *increases*, which contradicts with the distribution of $Y(x)$ illustrated in Figure 7(b). Consequently, our assumption is invalid, and point p is definitely closer to any point along q than p'. The proof completes. □

C. PROOF OF LEMMA 3.2

PROOF. Without loss of generality, we assume that there is at least one point s' along the interval $R_i \subseteq q$ such that $||p, s'|| < ||p_i, s'||$. As s' is a point on R_i, $||p, s'|| \geq dist(p, s') \geq mindist(p, q)$. On the other hand, $||p_i, s'|| = ||p_i, cp_i|| + dist(cp_i, s')$. Since cp_i is the control point of p_i over $R_i \subseteq q$, it is *visible* to any point along R_i. Thus, $dist(cp_i, s') \leq MAX(dist(cp_i, R_i.l), dist(cp_i, R_i.r))$, i.e., $||p_i, s'|| = ||p_i, cp_i|| + dist(cp_i, s') \leq ||p_i, cp_i|| + MAX(dist(cp_i, R_i.l), dist(cp_i, R_i.r)) = MAX(||p_i, R_i.l||, ||p_i, R_i.r||) \leq RL_{MAX} < mindist(p, q) \leq ||p, s'||$, which contradicts our assumption. The proof completes. □

D. PROOF OF THEOREM 4.1

PROOF. As illustrated in Figure 27, suppose there is at least one point s' on q such that the shortest path $SP(p, s')$ from point p to s' passes at least one vertex v of an obstacle o that is located *outside* the range $SR_{p,q}$, i.e., $v \in SP(p, s') \wedge v \in o \wedge mindist(o, q) > d$. Thus, $|SP(p, s')| = ||p, v|| + ||v, s'|| \geq dist(p, v) + dist(v, s')$. As $mindist(o, q) > d$, p locates inside $SR_{p,q}$, and s' lies on q, $dist(p, v) \geq d - mindist(p, q)$ and $dist(v, s') \geq mindist(o, q) >$

d. Hence, $|SP(p, s')| \geq dist(p, v) + dist(v, s') > d - mindist(p, q) + d = 2d - mindist(p, q)$, i.e., $|SP(p, s')| > 2d - mindist(p, q)$. On the other hand, since the shortest paths $SP(p, s)$ and $SP(p, e)$ are available and the query line segment q does not intersect any obstacle, there are two other obstacle-free paths from p to s', including (i) path $P_1(p, s')$ that follows $SP(p, s)$ and then from s to s', i.e., path $P_1(p, s') = \{v_1, v_2, s\}$, and (ii) path $P_2(p, s')$ that follows $SP(p, e)$ and then from e to s', i.e., path $P_2(p, s') = \{v_3, e\}$, as shown in Figure 27. Therefore, $|SP(p, s')| \leq \text{MIN}(|P_1(p, s')|, |P_2(p, s')|) = \text{MIN}(||p, s|| + dist(s, s'), ||p, e|| + dist(e, s')) \leq \text{MIN}(||p, s||, ||p, e||) + |q| = 2d - mindist(p, q)$, i.e., $|SP(p, s')| \leq 2d - mindist(p, q)$, which contradicts the above fact that $|SP(p, s')| > 2d - mindist(p, q)$. Consequently, our assumption is invalid, i.e., the path from p to s' via a vertex v of the obstacle o with $mindist(o, q) > d$ is not the shortest path. The proof completes. □

E. PROOF OF LEMMA 4.1

PROOF. If $P_2(p, s')$ is not the real shortest path $SP(p, s')$ from p to s', there must be another one $P_3(p, s') = SP(p, s')$ with $|P_3(p, s')| < |P_2(p, s')|$. Since $P_2(p, s')$ is the shortest one among all the paths from p to s' that only pass the vertexes of obstacles inside S_o, $P_3(p, s')$ must pass at least one vertex, denoted as v, of a certain obstacle o that is not included in S_o, i.e., $dist(v, s') \geq mindist(v, q) \geq mindist(o, q) > |P(p, s')|$. We further decompose $P_3(p, s')$ into two paths via node v, i.e., $P_3(p, v)$ and $P_3(v, s')$. As $|P_3(p, s')| = |P_3(p, v)| + |P_3(v, s')|$, $|P_3(p, s')| > |P_3(v, s')| \geq dist(v, s') \geq mindist(o, q) > |P(p, s')|$. On the other hand, $|P_2(p, s')| \leq |P(p, s')| < |P_3(p, s')|$. Therefore, $P_3(p, s')$ could not be the shortest path from p to s', and the proof completes. □

F. PROOF OF LEMMA 4.2

PROOF. As shown in Figure 8(a), suppose v is the control point of p over at least one point x on the interval $(VR_{u,q} \cap VR_{v,q})$. Since x is visible to both u and v, let path $P_1(p, x)$ be the shortest path from p to x via v, and $P_2(p, x)$ be the obstacle free path from p to x via u. We have $||p, x|| = |P_1(p, x)| = ||p, v|| + dist(v, x) = ||p, u|| + dist(u, v) + dist(v, x) > ||p, u|| + dist(u, x) = |P_2(p, x)|$, which contradicts the assumption that $P_1(p, x)$ is the shortest path. Thus, our assumption is invalid, and the proof completes. □

G. PROOF OF LEMMA 4.3

PROOF. Take $R = [s_3, s_4] \subseteq (VR_{v,q} - VR_{u,q}) = \{[s_1, s_2], [s_3, s_4]\}$ depicted in Figure 8(b) as an example. Here, point o_2 blocks u over R. Suppose point v satisfies all three conditions, i.e., (i) $v \notin \Delta us_3s_4$, (ii) $\forall x \in R$, segment $[v, x]$ intersects segment $[u, s_3]$, and (iii) endpoint s_3 is invisible to u because of the vertex m of obstacle o_2 and m is visible to any point along R. We assume v, although satisfying all three conditions, is the control point of point p for at least one point x on R. In other words, the shortest path from p to x must pass v, and we further assume $P_1(p, x)$ is the shortest path from p to x via u and v. Let y be the intersection between the segments $[v, x]$ and $[u, s_3]$, as shown in Figure 8(b). Therefore, we have $||p, x|| = |P_1(p, x)| = ||p, v|| + dist(v, x) = ||p, u|| + dist(u, v) + dist(v, x) = ||p, u|| + dist(u, v) + dist(v, y) + dist(y, x)$. On the other hand, for triangle Δuvy, $dist(u, v) + dist(v, y) > dist(u, y)$ holds. Thus, $|P_1(p, x)| > ||p, u|| + dist(u, y) + dist(y, x)$ holds. As $R \subseteq VR_{v,q}$, x is visible to v and point y must locate between m and s_3, i.e., $dist(u, y) + dist(y, x) = dist(u, m) + dist(m, y) + dist(y, x) > dist(u, m) + dist(m, x)$. Hence, $|P_1(p, x)| > ||p, u|| + dist(u, m) + dist(m, x)$ holds. Let $P_2(u, x)$ be an obstacle free path from u to x only via m, i.e., $|P_2(u, x)| = dist(u, m) + dist(m, x)$. Consequently, $|P_1(p, x)| > ||p, u|| + |P_2(u, x)|$ holds, which means that $P_1(p, x)$ is not the shortest path because its length is longer than an

obstacle free path from p to x via u and m. Thus, our assumption that $P_1(p, x)$ is not the shortest path from p to x via u and v is invalid. The proof completes. □

H. PROOF OF LEMMA 4.4

PROOF. If a node v in VG is contained in $CPL_{p,q}$, there must be at least one point s' on q such that the shortest path $SP(p, s')$ from p to s' passes v and s' is visible to v. We denote this shortest path (i.e., $SP(p, s')$) as $P_1(p, s')$ with $|SP(p, s')| = |P_1(p, s')| = ||p, v|| + dist(v, s') \geq ||p, v|| + mindist(v, q) \geq CPL_{MAX}$. On the other hand, let $\langle cp_i, R_i \rangle \in CPL_{p,q}$ be the tuple in $CPL_{p,q}$ such that s' is on $R_i = [R_i.l, R_i.r]$ ($\subseteq q$), and $P_2(p, s')$ be the path from p to s' via cp_i. Hence, $|P_2(p, s')| = ||p, cp_i|| + dist(cp_i, s')$. As $dist(cp_i, s') \leq \text{MAX}(dist(cp_i, R_i.l), dist(cp_i, R_i.r))$, $|P_2(p, s')| \leq ||p, cp_i|| + \text{MAX}(dist(cp_i, R_i.l), dist(cp_i, R_i.r)) = \text{MAX}(||p, cp_i|| + dist(cp_i, R_i.l), ||p, cp_i|| + dist(cp_i, R_i.r)) \leq \text{MAX}_{i\in[1, m]}(||p, cp_i|| + dist(cp_i, R_i.l), ||p, cp_i|| + dist(cp_i, R_i.r)) = CPL_{MAX} \leq |P_1(p, s')|$, i.e., $|P_2(p, s')| \leq |P_1(p, s')|$. Thus, $P_1(p, s')$ cannot be the shortest path from p to s', which contradicts our assumption. The proof completes. □

I. PROOF OF LEMMA 4.5

PROOF. The proof is straightforward because (i) data points in a given data set P will be evaluated in an incremental manner during the search unless the currently evaluated point satisfies the early termination condition (presented in Lemma 3.2); and (ii) only obstacles in a specified obstacle set O that may affect the obstructed distances between the current data point p processed and any point along a given query line segment q, i.e., the obstacles in O bounded by the search range $SR_{p,q}$ (defined in Theorem 4.1), are visited incrementally. □

J. PROOF OF LEMMA 4.6

PROOF. As shown in Algorithm 4, the CONN algorithm traverses the data R-tree T_p once in a best-first fashion to evaluate every data point in P that does not satisfy the early termination condition (presented in Lemma 3.2). In addition, it only traverses the obstacle R-tree T_o once as well. Although the IOR algorithm is invoked every time a new data point $p \in P$ is evaluated, it utilizes the obstacle vertexes preserved in the current visibility graph VG and traverses T_o in an incremental manner. □

K. PROOF OF THEOREM 4.2

PROOF. First, no answer points are missed (i.e., no false negatives) as each data point in P is accessed until the data point satisfying the early termination condition (presented in Lemma 3.2) is encountered. Second, the impact of each visited data point on the current result list RL is evaluated, which ensures no false positives (i.e., no false hits). □

L. PROOF OF LEMMA 5.1

PROOF. The proof is similar to the proof of Lemma 3.2 and hence omitted. □

M. PROOF OF LEMMA 6.1

PROOF. Suppose there is a point s' on q such that at least one vertex g along its shortest path to p, i.e., $SP(p, s')$, passes an obstacle o *outside* $ASR_{p,q}$, i.e., $\exists\, g \in SP(p, s')$, $g \in o \wedge o \cap ASR_{p,q} = \emptyset$. As s' is on q (which is one boundary of $ASR_{p,q}$) and $o \cap ASR_{p,q} = \emptyset$, $SP(p, s')$ must intersect the boundary of $ASR_{p,q}$, and let point x be one intersection. Without loss of generality, x could be located at $SP(p, s)$ or $SP(p, e)$ or q. If x is located at $SP(p, s)$, we have two paths from p to x, i.e., $P_1(p, x)$ following $SP(p, s')$ and $P_2(p, x)$

following $SP(p, s)$, but both stop at x instead of s'/s. Take Figure 15 as an example. $SP(p, s') = \{v_5, v_6\}$ and $SP(p, s) = \{v_1, v_2\}$. Correspondingly, $P_1(p, x) = \{v_5, v_6\}$ and $P_2(p, x) = \{v_1, v_2\}$. If $|P_1(p, x)| < |P_2(p, x)|$, $|P_1(p, x)| + ||x, s|| < |P_2(p, x)| + ||x, s|| = |SP(p, s)|$, which contradicts the fact that $SP(p, s)$ is the shortest path from p to s. Otherwise, $|P_1(p, x)| \geq |P_2(p, x)|$, i.e., $|P_1(p, x)| + ||x, s'|| \geq |P_2(p, x)| + ||x, s'||$; and thus our assumption that the path from p to s' passing vertex g is the shortest path is invalid. Similarly, we can prove that when x is located at $SP(p, e)$, the path from p to s' via an obstacle *outside* $ASR_{p,q}$ could not be the shortest path $SP(p, s')$. Since it is more likely that x is located at $SP(p, s)$ or $SP(p, e)$ but not q, we could conclude that *most*, but *not all*, of the obstacles that may affect the obstructed distance from p to any point along q should overlap the range $ASR_{p,q}$. The proof completes. □

N. PROOF OF LEMMA 6.2

PROOF. Suppose there is an obstacle o' such that (i) $o' \cap ASR_{p,q} \neq \emptyset$, (ii) $mindist(o', q) > \delta$, and (iii) there is at least one vertex v of o' located inside $ASR_{p,q}$, as shown in Figure 15. Let s'' be the projection of v on q, and p' be the intersection between a ray from s'' to v and the boundary of $ASR_{p,q}$ (i.e., either $SP(p, s)$ or $SP(p, e)$), as illustrated in Figure 15. Obviously, $dist(v, s'') \leq dist(p', s'') \leq \delta = \text{MAX}_{\forall v \in V_p}(mindist(v, q))$. On the other hand, since v is a vertex of the obstacle o' and s'' is v's projection on q, $mindist(o', q) \leq dist(v, s'')$. Thus, $mindist(o', q) \leq \delta$, which contradicts our assumption. The proof completes. □