

Spatial Queries in the Presence of Obstacles

Jun Zhang, Dimitris Papadias, Kyriakos Mouratidis, Manli Zhu

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
`{zhangjun, dimitris, kyriakos, cszhuml}@cs.ust.hk`

Abstract. Despite the existence of obstacles in many database applications, traditional spatial query processing utilizes the Euclidean distance metric assuming that points in space are directly reachable. In this paper, we study spatial queries in the presence of obstacles, where the obstructed distance between two points is defined as the length of the shortest path that connects them without crossing any obstacles. We propose efficient algorithms for the most important query types, namely, range search, nearest neighbors, e-distance joins and closest pairs, considering that both data objects and obstacles are indexed by R-trees. The effectiveness of the proposed solutions is verified through extensive experiments.

1. Introduction

This paper presents the first comprehensive approach for spatial query processing in the presence of obstacles. As an example of an "obstacle nearest neighbor query" consider Fig. 1 that asks for the closest point of q , where the definition of distance must now take into account the existing obstacles (shaded areas). Although point a is closer in terms of Euclidean distance, the actual nearest neighbor is point b (i.e., it is closer in terms of the *obstructed distance*). Such a query is typical in several scenarios, e.g., q is a pedestrian looking for the closest restaurant and the obstacles correspond to buildings, lakes, streets without crossings etc. The same concept applies to any spatial query, e.g., range search, spatial join and closest pair.

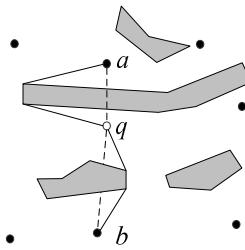


Fig. 1. An obstacle nearest neighbor query example

Despite the lack of related work in the Spatial Database literature, there is a significant amount of research in the context of Computational Geometry, where the problem is to devise main-memory, shortest path algorithms that take obstacles into account (e.g., find the shortest path from point a to b that does not cross any obstacle). Most existing approaches (reviewed in Section 2) construct a *visibility graph*, where each node corresponds to an obstacle vertex and each edge connects two vertices that are not obstructed by any obstacle. The algorithms pre-suppose the maintenance of the entire visibility graph in main memory. However, in our case this is not feasible due to the extreme space requirements for real spatial datasets. Instead we maintain local visibility graphs only for the obstacles that may influence the query result (e.g., for obstacles around point q in Fig. 1).

In the data clustering literature, *cod-clarans* [THH01] clusters objects into the same group with respect to the obstructed distance using the visibility graph, which is pre-computed and materialized. In addition to the space overhead, materialization is unsuitable for large spatial datasets due to potential updates in the obstacles or data (in which case a large part or the entire graph has to be re-reconstructed). Estivill-Castro and Lee [EL01] discuss several approaches for incorporating obstacles in spatial clustering. Despite some similarities with the problem at hand (e.g., visibility graphs), the techniques for clustering are clearly inapplicable to spatial query processing.

Another related topic regards query processing in spatial network databases [PZMT03], since in both cases movement is restricted (to the underlying network or by the obstacles). However, while obstacles represent areas where movement is prohibited, edges in spatial networks explicitly denote the permitted paths. This fact necessitates different query processing methods for the two cases. Furthermore, the target applications are different. The typical user of a spatial network database is a driver asking for the nearest gas station according to driving distance. On the other hand, the proposed techniques are useful in cases where movement is allowed in the whole data space except for the stored obstacles (vessels navigating in the sea, pedestrians walking in urban areas). Moreover, some applications may require the integration of both spatial network and obstacle processing techniques (e.g., a user that needs to find the best parking space near his destination, so that the sum of travel and walking distance is minimized).

For the following discussion we assume that there is one or more datasets of *entities*, which constitute the points of interest (e.g., restaurants, hotels) and a single obstacle dataset. The extension to multiple obstacle datasets or cases where the entities also represent obstacles is straightforward. Similar to most previous work on spatial databases, we assume that the entity and the obstacle datasets are indexed by R-trees [G84, SRF87, BKSS90], but the methods can be applied with any data partition index. Our goal is to provide a complete set of algorithms covering all common query types. The rest of the paper is organized as follows: Section 2 surveys the previous work focusing on directly related topics. Sections 3, 4, 5 and 6 describe the algorithms for range search, nearest neighbors, e -distance joins and closest pairs, respectively. Section 7 provides a thorough experimental evaluation and Section 8 concludes the paper with some future directions.

2. Related Work

Sections 2.1 and 2.2 discuss query processing in conventional spatial databases and spatial networks, respectively. Section 2.3 reviews obstacle path problems in main memory, and describes algorithms for maintaining visibility graphs. Section 2.4 summarizes the existing work and identifies the links with the current problem.

2.1 Query Processing in the Euclidean Space

For the following examples we use the R-tree of Fig. 2, which indexes a set of points $\{a, b, \dots, k\}$, assuming a capacity of three entries per node. Points that are close in space (e.g., a and b) are clustered in the same leaf node (N_3), represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root. R-trees (like most spatial access methods) were motivated by the need to efficiently process *range queries*, where the range usually corresponds to a rectangular window or a circular area around a query point. The R-tree answers the range query q (shaded area) in Fig. 2 as follows. The root is first retrieved and the entries (e.g., E_1, E_2) that intersect the range are recursively searched because they may contain qualifying points. Non-intersecting entries (e.g., E_4) are skipped. Notice that for non-point data (e.g., lines, polygons), the R-tree provides just a *filter* step to prune non-qualifying objects. The output of this phase has to pass through a *refinement* step that examines the actual object representation to determine the actual result. The concept of filter and refinement steps applies to all spatial queries on non-point objects.

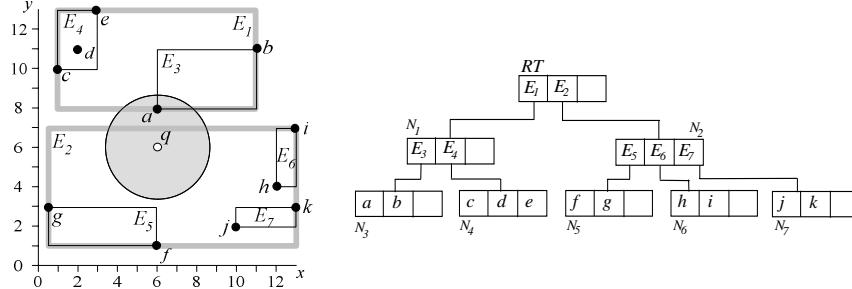


Fig. 2. An R-tree example

A *nearest neighbor* (NN) query retrieves the k ($k \geq 1$) data point(s) closest to a query point q . The R-tree NN algorithm proposed in [HS99] keeps a *heap* with the entries of the nodes visited so far. Initially, the heap contains the entries of the root sorted according to their minimum distance (*mindist*) from q . The entry with the minimum *mindist* in the heap (E_2 in Fig. 2) is expanded, i.e., it is removed from the heap and its children (E_5, E_6, E_7) are added together with their *mindist*. The next entry visited is E_1 (its *mindist* is currently the minimum in the heap), followed by E_3 , where the actual 1NN result (a) is found. The algorithm terminates, because the *mindist* of all entries in the heap is greater than the distance of a . The algorithm can be easily extended for the

retrieval of k nearest neighbors (k NN). Furthermore, it is optimal (it visits only the nodes necessary for obtaining the nearest neighbors) and *incremental*, i.e., it reports neighbors in ascending order of their distance to the query point, and can be applied when the number k of nearest neighbors to be retrieved is not known in advance.

The *e-distance join* finds all pairs of objects (s,t) $s \in S$, $t \in T$ within (Euclidean) distance e from each other. If both datasets S and T are indexed by R-trees, the *R-tree join* algorithm [BKS93] traverses synchronously the two trees, following entry pairs if their distance is below (or equal to) e . The *intersection join*, applicable for region objects, retrieves all intersecting object pairs (s,t) from two datasets S and T . It can be considered as a special case of the *e-distance join*, where $e=0$. Several spatial join algorithms have been proposed for the case where only one of the inputs is indexed by an R-tree or no input is indexed.

A closest-pairs query outputs the k ($k \geq 1$) pairs of points (s,t) $s \in S$, $t \in T$ with the smallest (Euclidean) distance. The algorithms for processing such queries [HS98, CMTV00] combine spatial joins with nearest neighbor search. In particular, assuming that both datasets are indexed by R-trees, the trees are traversed synchronously, following the entry pairs with the minimum distance. Pruning is based on the mindist metric, but this time defined between entry MBRs. Finally, a distance semi-join returns for each point $s \in S$ its nearest neighbor $t \in T$. This type of query can be answered either (i) by performing a NN query in T for each object in S , or (ii) by outputting closest pairs incrementally, until the NN for each entity in S is retrieved.

2.2 Query Processing in Spatial Networks

Papadias et al. [PZMT03] study the above query types for spatial network databases, where the network is modeled as a graph and stored as adjacency lists. Spatial entities are independently indexed by R-trees and are mapped to the nearest edge during query processing. The *network distance* of two points is defined as the distance of the shortest path connecting them in the graph. Two frameworks are proposed for pruning the search space: *Euclidean restriction* and *network expansion*.

Euclidean restriction utilizes the Euclidean *lower-bound property* (i.e., the fact that the Euclidean distance is always smaller or equal to the network distance). Consider, for instance, a range query that asks for all objects within network distance e from point q . The Euclidean restriction method first performs a conventional range query at the entity dataset and returns the set of objects S' within (Euclidean) distance e from q . Given the Euclidean lower bound property, S' is guaranteed to avoid false misses. Then, the network distance of all points of S' is computed and false hits are eliminated. Similar techniques are applied to the other query types, combined with several optimizations to reduce the number of network distance computations.

The network expansion framework performs query processing directly on the network without applying the Euclidean lower bound property. Consider again the example network range query. The algorithm first expands the network around the query point and finds all edges within range e from q . Then, an intersection join algorithm retrieves the entities that fall on these edges. Nearest neighbors, joins and closest pairs are processed using the same general concept.

2.3 Obstacle Path Problems in Main Memory

Path problems in the presence of obstacles have been extensively studied in Computational Geometry [BKOS97]. Given a set O of non-overlapping obstacles (polygons) in 2D space, a starting point p_{start} and a destination p_{end} , the goal is to find the shortest path from p_{start} to p_{end} which does not cross the interior of any obstacle in O . Fig. 3a shows an example where O contains 3 obstacles. The corresponding *visibility graph* G is depicted in Fig. 3b. The vertices of all the obstacles in O , together with p_{start} and p_{end} constitute the nodes of G . Two nodes n_i and n_j in G are connected by an edge if and only if they are mutually visible (i.e., the line segment connecting n_i and n_j does not intersect any obstacle interior). Since obstacle edges (e.g., n_1n_2) do not cross obstacle interiors, they are also included in G .

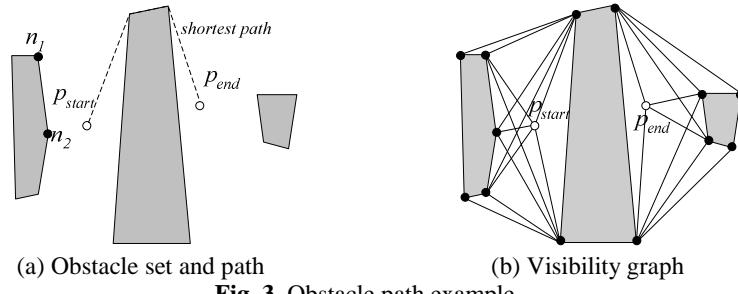


Fig. 3. Obstacle path example

It can be shown [LW79] that the shortest path contains only edges of the visibility graph. Therefore, the original problem can be solved by: (i) constructing G and (ii) computing the shortest path between p_{start} and p_{end} in G . For the second task any conventional shortest path algorithm [D59, KHI+86] suffices. Therefore, the focus has been on the first problem, i.e., the construction of the visibility graph. A naïve solution is to consider every possible pair of nodes in G and check if the line segment connecting them intersects the interior of any obstacle. This approach leads to $O(n^3)$ running time, where n is the number of nodes in G . In order to reduce the cost, Sharir and Schorr [SS84] perform a rotational plane-sweep for each graph node and find all the other nodes that are visible to it with total cost $O(n^2 \log n)$.

Subsequent techniques for visibility graph construction involve sophisticated data structures and algorithms, which are mostly of theoretical interest. The worst case optimal algorithm [W85, AGHI86] performs a rotational plane-sweep for all the vertices simultaneously and runs in $O(n^2)$ time. The optimal output-sensitive approaches [GM87, R95, PV96] have $O(m+n\log n)$ running time, where m is the number of edges in G . If all obstacles are convex, it is sufficient to consider the *tangent visibility graph* [PV95], which contains only the edges that are tangent to two obstacles.

2.4 Discussion

In the rest of the paper we utilize several of these findings for efficient query processing. First the Euclidean *lower-bound property* also holds in the presence of obstacles,

since the Euclidean distance is always smaller or equal to the obstructed distance. Thus, the algorithms of Section 2.1 can be used to return a set of candidate entities, which includes the actual output, as well as, a set of *false hits*. This is similar to the Euclidean restriction framework for spatial networks, discussed in Section 2.2. The difference is that now we have to compute the obstructed (as opposed to network) distances of the candidate entities. Although we take advantage of visibility graphs to facilitate obstructed distance computation, in our case it is not feasible to maintain in memory the complete graph due to the extreme space requirements for real spatial datasets. Furthermore, pre-materialization is unsuitable for updates in the obstacle or entity datasets. Instead we construct visibility graphs on-line, taking into account only the obstacles and the entities relevant to the query. In this way, updates in individual datasets can be handled efficiently, new datasets can be incorporated in the system easily (as new information becomes available), and the visibility graph is kept small (so that distance computations are minimized).

3. Obstacle Range Query

Given a set of obstacles O , a set of entities P , a query point q and a range e , an obstacle range (OR) query returns all the objects of P that are within obstructed distance e from q . The OR algorithm processes such a query as follows: (i) it first retrieves the set P' of candidate entities that are within Euclidean distance e (from q) using a conventional range query on the R-tree of P ; (ii) it finds the set O' of obstacles that are relevant to the query; (iii) it builds a local visibility graph G' containing the elements of P' and O' ; (iv) it removes false hits from P' by evaluating the obstructed distance for each candidate object using G' . Consider the example OR query q (with $e = 6$) in Fig. 4a, where the shaded areas represent obstacles and points correspond to entities.

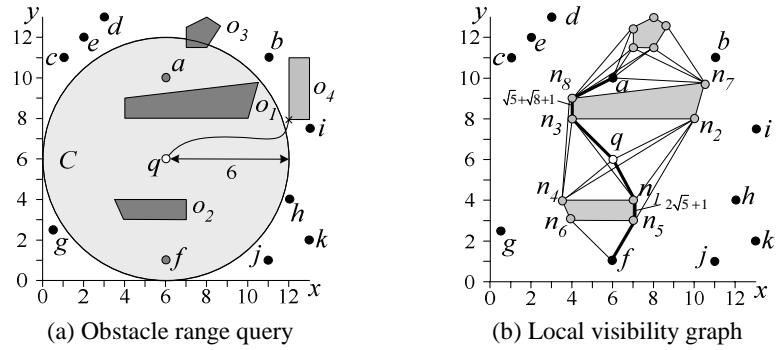


Fig. 4. Example of obstacle range query

Clearly, the set P' of entities intersecting the disk C centered at q with radius e , constitutes a superset of the query result. In order to remove the false hits we need to retrieve the relevant obstacles. A crucial observation is that only the obstacles intersecting C may influence the result. By the Euclidean lower-bound property, any path that

starts from q and ends at any vertex of an obstacle that lies outside C (e.g., curve in Fig. 4a), has length larger than the range e . Therefore, it is safe to exclude the obstacle (o_4) from the visibility graph. Thus, the set O' of relevant obstacles can be found using a range query (centered at q with radius e) on the R-tree of O . The local visibility graph G' for the example of Fig. 4a is shown in Fig. 4b. For constructing the graph, we use the algorithm of [SS84], without tangent simplification.

The final step evaluates the obstructed distance between q and each candidate. In order to minimize the computation cost, OR expands the graph around the query point q only once for all candidate points using a traversal method similar to the one employed by Dijkstra's algorithm [D59]. Specifically, OR maintains a priority queue Q , which initially contains the neighbors of q (i.e., n_1 to n_4 in Fig. 4b) sorted by their obstructed distance. Since these neighbors are directly connected to q , the obstructed distance $d_O(n_i, q)$, $1 \leq i \leq 4$, equals the Euclidean distance $d_E(n_i, q)$. The first node (n_1) is dequeued and inserted into a set of visited nodes V . For each unvisited neighbor n_x of n_1 (i.e., $n_x \notin V$), $d_O(n_x, q)$ is computed, using n_1 as an intermediate node i.e., $d_O(n_x, q) = d_O(n_1, q) + d_E(n_1, n_x)$. If $d_O(n_x, q) \leq e$, n_x is inserted in Q . Fig. 5 illustrates the OR algorithm.

```

Algorithm OR( $RT_p$ ,  $RT_o$ ,  $q$ ,  $e$ )
/*  $RT_p$  is the entity R-tree,  $RT_o$  is the obstacle R-tree,  $q$  is the
query point,  $e$  is the query range */
 $P' = \text{Euclidean\_range}(RT_p, q, e)$  // get qualifying entities
 $O' = \text{Euclidean\_range}(RT_o, q, e)$  // get relevant obstacles
 $G' = \text{build\_visibility\_graph}(q, P', O')$  // algorithm of [SS84]
 $V = \emptyset$ ;  $R = \emptyset$  //  $V$  is the set of visited nodes,  $R$  is the result
insert  $\langle q, 0 \rangle$  into  $Q$ 
while  $Q$  and  $P'$  are both non-empty
    de-queue  $\langle n, d_o(n, q) \rangle$  from  $Q$  //  $n$  has the min  $d_o(n, q)$ 
    if  $n \notin V$  //  $n$  is an unvisited node
        if  $n \in P'$  //  $n$  is an unreported entity
             $R = R \cup \{n\}$ ;  $P' = P' - \{n\}$ 
            for each neighbor node  $n_x$  of  $n$ 
                if  $(n_x \notin V)$ 
                     $d_o(n_x, q) = d_o(n, q) + d_E(n, n_x)$ 
                    if  $(d_o(n_x, q) \leq e)$ 
                        insert  $\langle n_x, d_o(n_x, q) \rangle$  into  $Q$ 
 $V = V \cup n$ 
return  $R$ 
End OR

```

Fig. 5. OR algorithm

Note that it is possible for a node to appear multiple times in Q , if it is found through different paths. For instance, in Fig. 4b, n_2 may be re-inserted after visiting n_1 . Duplicate elimination is performed during the de-queuing process, i.e., a node is visited only the first time that it is de-queued (with the smallest distance from q). Subsequent visits are avoided by checking the contents of V (set of already visited nodes). When the de-queued node is an entity, it is reported and removed from P' . The algorithm terminates when the queue or P' is empty.

4. Obstacle Nearest Neighbor Query

Given a query point q , an obstacle set O and an entity set P , an obstacle nearest neighbor (ONN) query returns the k objects of P that have the smallest obstructed distances from q . Assuming, for simplicity, the retrieval of a single neighbor ($k=1$) in Fig. 6, we illustrate the general idea of ONN algorithm before going into details. First the Euclidean nearest neighbor of q (object a) is retrieved from P using an incremental algorithm (e.g., [HS99] in Section 2.1) and $d_O(a,q)$ is computed. Due to the Euclidean lower-bound property, objects with potentially smaller obstructed distance than a should be within Euclidean distance $d_{E_{max}} = d_O(a,q)$. Then, the next Euclidean neighbor (f) within the $d_{E_{max}}$ range is retrieved and its obstructed distance is computed. Since $d_O(f,q) < d_O(a,q)$, f becomes the current NN and $d_{E_{max}}$ is updated to $d_O(f,q)$ (i.e., $d_{E_{max}}$ continuously shrinks). The algorithm terminates when there is no Euclidean nearest neighbor within the $d_{E_{max}}$ range.

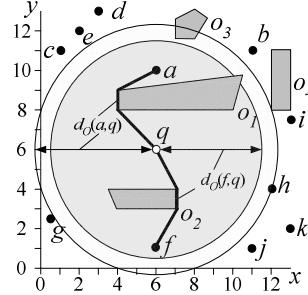


Fig. 6. Example of obstacle nearest neighbor query

It remains to clarify the obstructed distance computation. Consider, for instance, Fig. 7 where the Euclidean NN of q is point p . In order to compute $d_O(p,q)$, we first retrieve the obstacles o_1, o_2 within the range $d_E(p,q)$ and build an initial visibility graph that contains o_1, o_2, p and q . A provisional distance $d_{O_I}(p,q)$ is computed using a shortest path algorithm (we apply Dijkstra's algorithm). The problem is that the graph is not sufficient for the actual distance, since there may exist obstacles (o_3, o_4) outside the range that obstruct the shortest path from q to p .

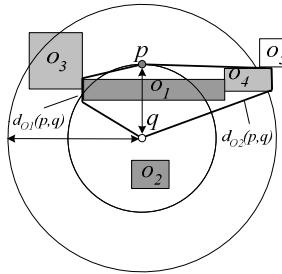


Fig. 7. Example of obstructed distance computation

In order to find such obstacles, we perform a second Euclidean range query on the obstacle R-tree using $d_{O1}(p,q)$ (i.e., the large circle in Fig. 7). The new obstacles o_3 and o_4 are added to the visibility graph, and the obstructed distance $d_{O2}(p,q)$ is computed again. The process has to be repeated, since there may exist another obstacle (o_5) outside the range $d_{O2}(p,q)$ that intersects the new shortest path from q to p . The termination condition is that there are no new obstacles in the last range, or equivalently, the shortest path remains the same in two subsequent iterations, meaning that the last set of added obstacles does not affect $d_O(p,q)$ (note that the obstructed distance can only increase in two subsequent iterations as new obstacles are discovered). The pseudo-code of the algorithm is shown in Fig. 8. The initial visibility graph G' , passed as a parameter, contains p , q and the obstacles in the Euclidean range $d_E(p,q)$.

```

Algorithm compute_obstructed_distance( $G$ ,  $p$ ,  $q$ ,  $G'$ ,  $RT_o$ )
 $d_o(p,q) = \text{shortest\_path\_dist}(G', p, q)$ 
 $O' = \text{set of obstacles in } G'$ 
repeat
     $O_{\text{new}} = \text{Euclidean\_range}(RT_o, q, d_o(p,q))$ 
    if  $O' \subset O_{\text{new}}$ 
        for each obstacle  $o$  in  $O_{\text{new}} - O'$ 
            add_obstacle( $o, G'$ )
         $d_o(p,q) = \text{shortest\_path\_dist}(G', p, q)$ 
         $O' = O_{\text{new}}$ 
    else // termination condition
        return  $d_o(p,q)$ 
End compute_obstructed_distance

```

Fig. 8. Obstructed distance computation

The final remark concerns the dynamic maintenance of the visibility graph in main memory. The following basic operations are implemented, to avoid re-building the graph from scratch for each new computation:

- *Add_obstacle*(o, G') is used by the algorithm of Fig. 8 for incorporating new obstacles in the graph. It adds all the vertices of o to G' as nodes and creates new edges accordingly. It removes existing edges that cross the interior of o .
- *Add_entity*(p, G') incorporates a new point in an existing graph. If, for instance, in the example of Fig. 7 we want the two nearest neighbors, we re-use the graph that we constructed for the 1st NN to compute the distance of the second one. The operation adds p to G' and creates edges connecting it with the visible nodes in G' .
- *Delete_entity*(p, G') is used to remove entities for which the distance computations have been completed.

Add obstacle performs a rotational plane-sweep for each vertex of o and adds the corresponding edges to G' . A list of all obstacles in G' is maintained to facilitate the sweep process. Existing edges that cross the interior of o are removed by an intersection check. *Add entity* is supported by performing a rotational plane-sweep for the newly added node to reveal all its edges. The *delete entity* operation just removes p and its incident edges.

Fig. 9 illustrates the complete algorithm for retrieval of k (≥ 1) nearest neighbors. The k Euclidean NNs are first obtained using the entity R-tree, sorted in ascending order of their obstructed distance to q , and $d_{E_{max}}$ is set to the distance of the k^{th} point. Similar to the single NN case, the subsequent Euclidean neighbors are retrieved incrementally, while maintaining the k (obstructed) NNs and $d_{E_{max}}$ (except that $d_{E_{max}}$

equals the obstructed distance of the k -th neighbor), until the next Euclidean NN has larger Euclidean distance than $d_{E_{max}}$.

```

Algorithm ONN( $RT_p$ ,  $RT_o$ ,  $q$ ,  $k$ )
/*  $RT_p$  is the entity R-tree,  $RT_o$  is the obstacle R-tree,  $q$  is the
query,  $k$  is number of NN requested */
 $R = \emptyset$  //  $R$  is the result
 $P' = \text{Euclidean\_NN}(RT_p, q, k)$ ; // find the  $k$  Euclidean NNs of  $q$ 
 $O' = \text{Euclidean\_range}(RT_o, q, d(p_k, q))$ 
 $G' = \text{build\_visibility\_graph}(q, P', O')$ 
for each entity  $p_i$  in  $P'$ 
     $d_o(p_i, q) = \text{compute\_obstructed\_distance}(G', p_i, q)$ 
     $\text{delete\_entity}(p_i, G')$ 
sort  $P'$  in ascending order of  $d_o(p_i, q)$  and insert into  $R$ 
 $d_{E_{max}} = d_o(p_k, q)$  //  $p_k$  is the farthest NN
repeat
    ( $p, d_e(p, q)$ ) =  $\text{next\_Euclidean\_NN}(RT_p, q)$ ;
     $\text{add\_entity}(p, G')$ 
     $d_o(p, q) = \text{compute\_obstructed\_distance}(G', p, q)$ 
     $\text{delete\_entity}(p, G')$ 
    if ( $d_o(p, q) < d_o(p_k, q)$ ) //  $p$  is closer than the  $k^{th}$  NN
         $R = R - \{p_k\}$ 
        insert  $p$  in  $R$  so that  $R$  remains sorted by  $d_o$ 
         $d_{E_{max}} = d_o(p_k, q)$  // update the Euclidean threshold
until  $d_e(p, q) > d_{E_{max}}$ 
return  $R$ 
End ONN

```

Fig. 9. ONN algorithm

5. Obstacle e -Distance Join

Given an obstacle set O , two entity datasets S, T and a value e , an obstacle e -distance join (ODJ) returns all entity pairs (s, t) , $s \in S$, $t \in T$ such that $d_O(s, t) \leq e$. Based on the Euclidean lower-bound property, the ODJ algorithm processes an obstacle e -distance join as follows: (i) it performs an Euclidean e -distance join on the R-trees of S and T to retrieve entity pairs (s, t) with $d_E(s, t) \leq e$; (ii) it evaluates $d_O(s, t)$ for each candidate pair (s, t) and removes false hits. The R-tree join algorithm [BKS93] (see Section 2.1) is applied for step (i). For step (ii) we use the obstructed distance computation algorithm of Fig. 8.

Observe that although the number of distance computations equals the cardinality of the Euclidean join, the number of applications of the algorithm can be significantly smaller. Consider, for instance, that the Euclidean join retrieves five pairs: (s_1, t_1) , (s_1, t_2) , (s_1, t_3) , (s_2, t_1) , (s_2, t_4) , requiring five obstructed distance computations. However, there are only two objects $s_1, s_2 \in S$ participating in the candidate pairs, implying that all five distances can be computed by building only two visibility graphs around s_1 and s_2 . Based on this observation, ODJ counts the number of distinct objects from S and T in the candidate pairs. The dataset with the smallest count is used to provide the 'seeds' for visibility graphs. Let Q be the set of points of the 'seed' dataset that appear in the Euclidean join result (i.e., in the above example $Q = \{s_1, s_2\}$). Similarly, P is the set of points of the second dataset that appear in the result (i.e., $P = \{t_1, t_2, t_3, t_4\}$). The problem can then be converted to: for each $q \in Q$ and a set $P' \subseteq P$ of candidates (paired with q in the Euclidean join), find the objects of P' that are within obstructed distance

e from q . This process corresponds to the false hit elimination part of the obstacle range query and can be processed by an algorithm similar to OR (Fig. 5). To exploit spatial locality between subsequent accesses to the obstacle R-tree (needed to retrieve the obstacles for the visibility graph for each range), ODJ sorts and processes the seeds by their Hilbert order. The pseudo code of the algorithm is shown in Fig. 10.

```

Algorithm ODJ( $RT_s$ ,  $RT_r$ ,  $RT_o$ ,  $e$ )
/*  $RT_s$  and  $RT_r$  is the entity R-trees,  $RT_o$  is the obstacle R-tree,  $e$  is
the query range */
 $R = \emptyset$ 
 $Rjoin-res = Euclidean\_distance\_join(RT_s, RT_r, e)$ 
compute  $Q$  and  $P$ ;
sort  $Q$  according to Hilbert order // to maximize locality
for each object  $q \in Q$ 
     $P' =$  set of objects  $\in P$  that appear with  $q$  in  $Rjoin-res$ 
     $O' = Euclidean\_range(RT_o, q, e)$  // get relevant obstacles
     $R' = OR(P', O', q, e)$  // eliminate false hits
     $R = R \cup \{<q, r> / r \in R'\}$ 
return  $R$ 
End ODJ

```

Fig. 10. ODJ algorithm

6. Obstacle Closest-Pair Query

Given an obstacle set O , two entity datasets S, T and a value $k \geq 1$, an obstacle closest-pair (OCP) query retrieves the k entity pairs (s, t) , $s \in S, t \in T$, that have the smallest $d_O(s, t)$. The OCP algorithm employs an approach similar to ONN. Assuming for example, that only the (single) closest pair is requested, OCP: (i) performs an incremental closest pair query on the entity R-trees of S and T and retrieves the Euclidean closest pair (s, t) ; (ii) it evaluates $d_O(s, t)$ and uses it as a bound $d_{E\max}$ for Euclidean closest-pairs search; (iii) it obtains the next closest pair (within Euclidean distance $d_{E\max}$), evaluates its obstructed distance and updates the result and $d_{E\max}$ if necessary; (iv) it repeats step (iii) until the incremental search for pairs exceeds $d_{E\max}$. Fig. 11 shows the OCP algorithm for retrieval of k closest-pairs.

```

Algorithm OCP( $RT_s$ ,  $RT_r$ ,  $RT_o$ ,  $k$ )
/*  $RT_s$  and  $RT_r$  is the entity R-trees,  $RT_o$  is the obstacle R-tree,  $k$  is
the number of pairs requested */
 $\{(s_1, t_1), \dots, (s_k, t_k)\} = Euclidean\_CP(RT_s, RT_r, k)$ 
sort  $(s_i, t_i)$  in ascending order of their  $d_o(s_i, t_i)$ 
 $d_{E\max} = d_o(s_k, t_k)$ 
repeat
     $(s', t') = next\_Euclidean\_CP(RT_s, RT_r)$ 
     $d_o(s', t') = compute\_obstructed\_distance(G', s', t')$ 
    if  $(d_o(s', t') < d_{E\max})$ 
        delete  $(s_k, t_k)$  from  $\{(s_1, t_1), \dots, (s_k, t_k)\}$  and insert  $(s', t')$  in it, so that it remains sorted by  $d_o$ 
         $d_{E\max} = d_o(s_k, t_k)$ 
    until  $d_o(s', t') > d_{E\max}$ 
return  $\{(s_1, t_1), \dots, (s_k, t_k)\}$ 
End OCP

```

Fig. 11. OCP algorithm

OCP first finds the k Euclidean pairs, it evaluates their obstructed distances and treats the maximum distance as $d_{E\max}$. Subsequent candidate pairs are retrieved incrementally, continuously updating the result and $d_{E\max}$ until no pairs are found within the $d_{E\max}$ bound. Note that the algorithm (and ONN presented in Section 4) is not suitable for *incremental* processing, where the value of k is not set in advance. Such a situation may occur if a user just browses through the results of a closest pair query (in increasing order of the pair distances), without a pre-defined termination condition. Another scenario where incremental processing is useful concerns complex queries: "find the city with more than 1M residents, which is closest to a nuclear factory". The output of the top-1 CP may not qualify the population constraint, in which case the algorithm has to continue reporting results until the condition is satisfied.

In order to process incremental queries we propose a variation of the OCP algorithm, called *iOCP* (for *incremental*), shown in Fig. 12 (note that now there is not a k parameter). When a Euclidean CP (s, t) is obtained, its obstructed distance $d_O(s, t)$ is computed and the entry $\langle (s, t), d_O(s, t) \rangle$ is inserted into a queue Q . The observation is that all the pairs (s_i, t_j) in Q such that $d_O(s_i, t_j) \leq d_E(s, t)$, can be immediately reported, since no subsequent Euclidean CP can lead to a lower obstructed distance. The same methodology can be applied for deriving an incremental version of ONN.

```

Algorithm iOCP( $RT_s$ ,  $RT_t$ ,  $RT_o$ )
repeat
     $(s, t) = \text{next\_Euclidean\_CP}(RT_s, RT_t)$ 
     $d_o(s, t) = \text{compute\_obstructed\_distance}(s, t)$ 
    insert  $\langle (s, t), d_o(s, t) \rangle$  into  $Q$ 
    for each  $(s_i, t_j)$  such that  $d_o(s_i, t_j) \leq d_o(s, t)$ 
        de-heap  $\langle (s_i, t_j), d_o(s_i, t_j) \rangle$  from  $Q$ 
        report  $\langle (s_i, t_j), d_o(s_i, t_j) \rangle$ 
until termination condition
return
End iOCP

```

Fig. 12. *iOCP* algorithm

7. Experiments

In this section, we experimentally evaluate the CPU time and I/O cost of the proposed algorithms, using a Pentium III 733MHz PC. We employ R*-trees [BKSS90], assuming a page size of 4K (resulting in a node capacity of 204 entries) and an LRU buffer that accommodates 10% of each R-tree participating in the experiments. The obstacle dataset contains $|O| = 131,461$ rectangles, representing the MBRs of streets in Los Angeles [Web] (but as discussed in the previous sections, our methods support arbitrary polygons). To control the density of the entities, the entity datasets are synthetic, with cardinalities ranging from $0.01 \cdot |O|$ to $10 \cdot |O|$. The distribution of the entities follows the obstacle distribution; the entities are allowed to lie on the boundaries of the obstacles but not in their interior. For the performance evaluation of the range and nearest neighbor algorithms, we execute workloads of 200 queries, which also follow the obstacle distribution.

7.1 Range Queries

First, we present our experimental results on obstacle range queries. Fig. 13a and Fig. 13b show the performance of the OR algorithm in terms of I/O cost and CPU time, as functions of $|P|/|O|$ (i.e., the ratio of entity to obstacle dataset cardinalities), fixing the query range e to 0.1% of the data universe side length. The I/O cost for entity retrieval increases with $|P|/|O|$ because the nodes that lie within the (fixed) range e in the entity R-tree grows with $|P|$. However, the page accesses for obstacle retrieval remain stable, since the number of obstacles that participate in the distance computations (i.e., the ones intersecting the range) is independent of the entity dataset cardinality. The CPU time grows rapidly with $|P|/|O|$, because the visibility graph construction cost is $O(n^2 \log n)$ and the value of n increases linearly with the number of entities in the range (note the logarithmic scale for CPU cost).

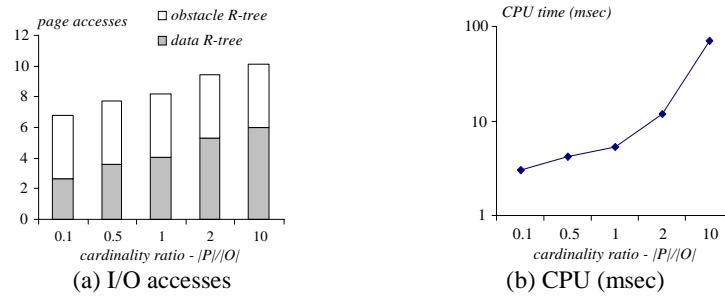


Fig. 13. Cost vs. $|P|/|O|$ ($e=0.1\%$)

Fig. 14 depicts the performance of OR as a function of e , given $|P|=|O|$. The I/O cost increases quadratically with e because the number of objects and nodes intersecting the Euclidean range is proportional to its area (which is quadratic with e). The CPU performance again deteriorates even faster because of the $O(n^2 \log n)$ graph construction cost.

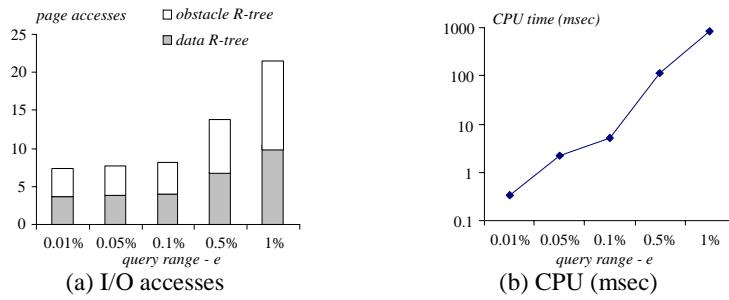


Fig. 14. Cost vs. e ($|P|=|O|$)

The next experiment evaluates the number of false hits, i.e., objects within the Euclidean, but not in the obstructed range. Fig. 15a shows the false hit ratio (number of false hits / number of objects in the obstructed range) for different cardinality ratios (fixing $e=0.1\%$), which remains almost constant (the absolute number of false hits increases

linearly with $|P|$). Fig. 15b shows the false hit ratio as a function of e (for $|P|=|O|$). For small e values, the ratio is low because the numbers of candidate entities and obstacles that obstruct their view is limited. As a result, the difference between Euclidean and obstructed distance is insignificant. On the other hand, the number of obstacles grows quadratically with e , increasing the number of false hits.

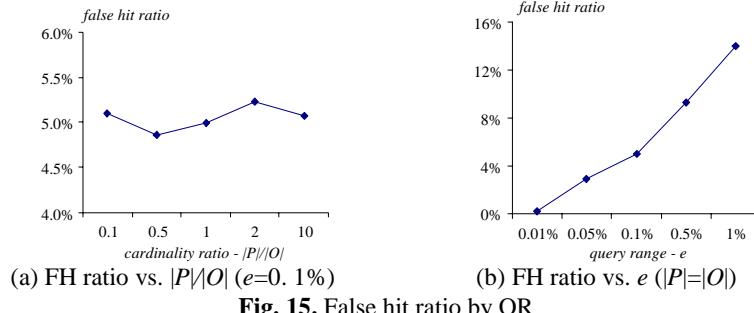


Fig. 15. False hit ratio by OR

7.2 Nearest Neighbor Queries

This set of experiments focuses on obstacle nearest neighbor queries. Fig. 16 illustrates the costs of the ONN algorithm as function of the ratio $|P||O|$, fixing the number k of neighbors to 16. The page accesses of the entity R-tree do not increase fast with $|P||O|$ because, as the density increases, the range around the query point where the Euclidean neighbors are found decreases. As a result the obstacle search radius (and the number of obstacles that participate in the obstructed distance computations) also declines. Fig. 16b confirms this observation, showing that the CPU time drops significantly with the data density.

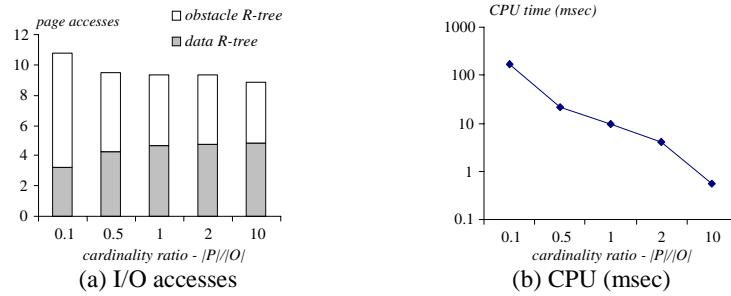


Fig. 16. Cost vs. $|P||O|$ ($k=16$)

Fig. 17 shows the performance of ONN for various values of k when $|P|=|O|$. As expected, both the I/O cost and CPU time of the algorithm grow with k , because a high value of k implies a larger range to be searched (for entities and obstacles) and more distance computations. Fig. 18a shows the impact of $|P||O|$ on the false hit ratio ($k = 16$). A relatively small cardinality $|P|$ results in large deviation between Euclidean and obstructed distances, therefore incurring high false hit ratio, which is gradually allevi-

ated as $|P|$ increases. In Fig. 18b we vary k and monitor the false hit ratio. Interestingly, the false hit ratio obtains its maximum value for $k \approx 4$ and starts decreasing when $k > 4$. This can be explained by the fact that, when k becomes high, the set of k Euclidean NN contains a big portion of the k actual (obstructed) NN, despite their probably different internal ordering (e.g., the 1st Euclidean NN is 3rd obstructed NN).

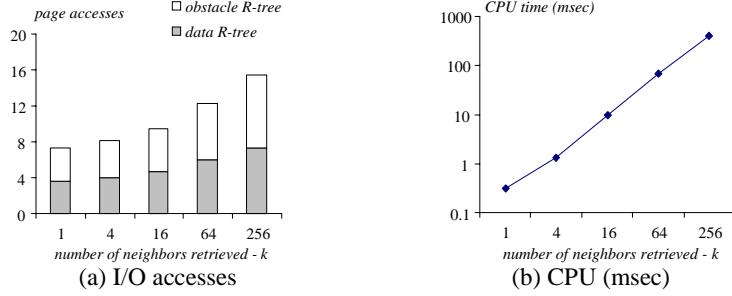


Fig. 17. Cost vs. k ($|P|=|O|$)

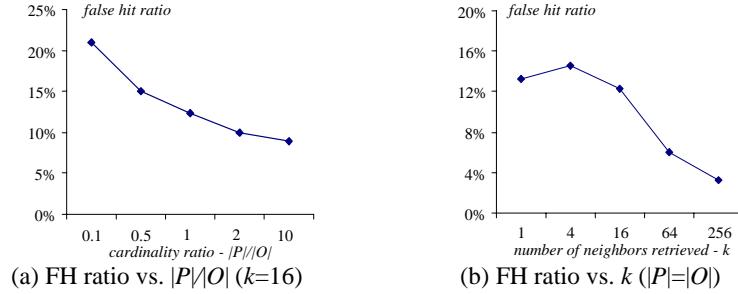


Fig. 18. False hit ratio by ONN

7.3 e -Distance Joins

We proceed with the performance study of the e -distance join algorithm, using $|T|=0.1|O|$ and setting the join distance e to 0.01% of the universe length. Fig. 19a plots the number of disk accesses as a function of $|S||O|$, ranging from 0.01 to 1. The number of page accesses for the entity R-trees grows much slower than the obstacle R-tree because the cost of the Euclidean join is not very sensitive to the density. On the other hand, the output size (of the Euclidean join) grows fast with the density, increasing the number of obstructed distance evaluations and the accesses to the obstacle R-tree (in the worst case each Euclidean pair initiates a new visibility graph). This observation is verified in Fig. 19b which shows the CPU cost as a function of $|S||O|$.

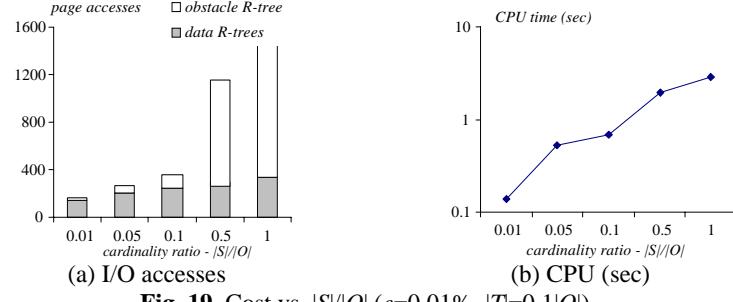


Fig. 19. Cost vs. $|S|/|O|$ ($e=0.01\%$, $|T|=0.1|O|$)

In Fig. 20a, we set $|S|=|T|=0.1|O|$ and measure the number of disk accesses for varying e . The page accesses for the entity R-tree do not have large variance (they range between 230 for $e = 0.001\%$ and 271 for $e = 0.1\%$) because the node extents are large with respect to the range. However, as in the case of Fig. 20a, the output of the Euclidean joins (and the number of obstructed distance computations) grows fast with e , which is reflected in the page accesses for the obstacle R-tree and the CPU time (Fig. 20b).

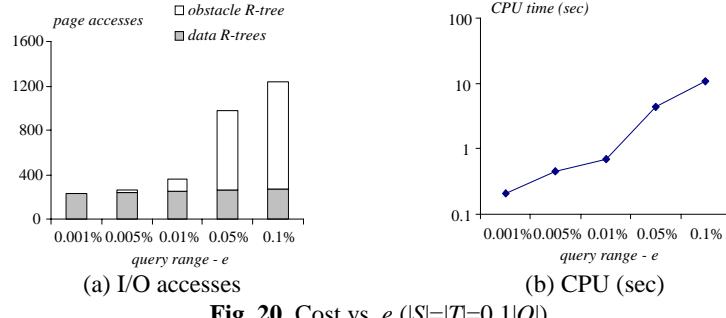


Fig. 20. Cost vs. e ($|S|=|T|=0.1|O|$)

7.4 Closest Pairs

Next, we evaluate the performance of closest pairs in the presence of obstacles. Fig. 21 plots the cost of the OCP algorithm as a function of $|S|/|O|$ for $k=16$ and $|T|=0.1|O|$. The I/O cost of the entity R-trees grows with the cardinality ratio (i.e., density of S), which is caused by the Euclidean closest-pair algorithm (similar observations were made in [CMTV00]). On the other hand, the density of S does not affect significantly the accesses to the obstacle R-tree because high density leads to closer distance between the Euclidean pairs. The CPU time of the algorithm (shown in Fig. 21b) grows fast with $|S|/|O|$, because the dominant factor is the computation required for obtaining the Euclidean closest pairs (as opposed to obstructed distances).

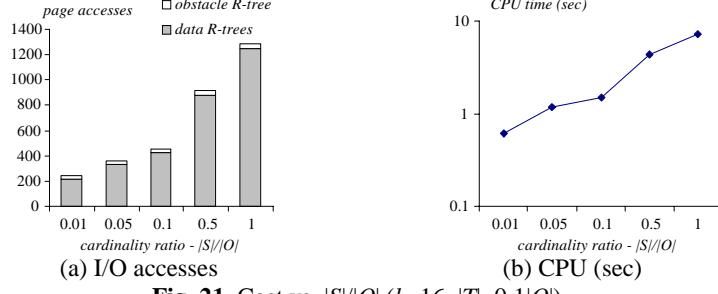


Fig. 21. Cost vs. $|S|/|O|$ ($k=16$, $|T|=0.1|O|$)

Fig. 22 shows the cost of the algorithm with $|S|=|T|=0.1|O|$ for different values of k . The page accesses for the entity R-trees (caused by the Euclidean CP algorithm) remain almost constant, since the major cost occurs before the first pair is output (i.e., the k closest pairs are likely to be in the heap after the first Euclidean NN is found, and are returned without extra IOs). The accesses to the obstacle R-tree and the CPU time, however, increase with k because more obstacles must be taken into account during the construction of the visibility graphs.

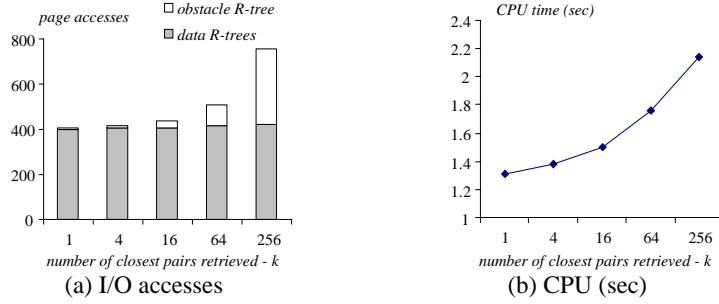


Fig. 22. Cost vs. k ($|S|=|T|=0.1|O|$)

8. Conclusion

This paper tackles spatial query processing in the presence of obstacles. Given a set of entities P and a set of polygonal obstacles O , our aim is to answer spatial queries with respect to the obstructed distance metric, which corresponds to the length of the shortest path that connects them without passing through obstacles. This problem has numerous important applications in real life, and several main memory algorithms have been proposed in Computational Geometry. Surprisingly, there is no previous work for disk-resident datasets in the area of Spatial Databases.

Combining techniques and algorithms from both aforementioned fields, we propose an integrated framework that efficiently answers most types of spatial queries (i.e., range search, nearest neighbors, e -distance joins and closest pairs), subject to obstacle

avoidance. Making use of local visibility graphs and effective R-tree algorithms, we present and evaluate a number of solutions. Being the first thorough study of this problem in the context of massive datasets, this paper opens a door to several interesting directions for future work. For instance, as objects move in practice, it would be interesting to study obstacle queries for moving entities and/or moving obstacles.

References

- [AGHI86] Asano, T., Guibas, L., Hershberger, J., Imai, H. Visibility of Disjoint Polygons. *Algorithmica* 1, 49-63, 1986.
- [BKOS97] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. *Computational Geometry*. pp. 305-315, Springer, 1997.
- [BKS93] Brinkhoff, T., Kriegel, H., Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *SIGMOD*, 1993.
- [BKSS90] Becker, B., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method. *SIGMOD*, 1990.
- [CMTV00] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M. Closest Pair Queries in Spatial Databases. *SIGMOD*, 2000.
- [D59] Dijkstra, E. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, 1, 269-271, 1959.
- [EL01] Estivill-Castro, V., Lee, I. Fast Spatial Clustering with Different Metrics in the Presence of Obstacles. *ACM GIS*, 2001.
- [G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.
- [GM87] Ghosh, S., Mount, D. An Output Sensitive Algorithm for Computing Visibility Graphs. *FOCS*, 1987.
- [HS98] Hjaltason, G., Samet, H. Incremental Distance Join Algorithms for Spatial Databases. *SIGMOD*, 1998.
- [HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *TODS*, 24(2), 265-318, 1999.
- [KHI+86] Kung, R., Hanson, E., Ioannidis, Y., Sellis, T., Shapiro, L., Stonebraker, M. Heuristic Search in Data Base Systems. *Expert Database Systems*, 1986.
- [LW79] Lozano-Pérez, T., Wesley, M. An Algorithm for Planning Collision-free Paths among Polyhedral Obstacles. *CACM*, 22(10), 560-570, 1979.
- [PV95] Pocchiola, M., Vegter, G. Minimal Tangent Visibility Graph. *Computational Geometry: Theory and Applications*, 1995.
- [PV96] Pocchiola, M., Vegter, G. Topologically Sweeping Visibility Complexes via Pseudo-triangulations. *Discrete Computational Geometry*, 1996.
- [PZMT03] Papadias, D., Zhang, J., Mamoulis, N., Tao, Y. Query Processing in Spatial Network Databases. *VLDB*, 2003.
- [R95] Rivière, S. Topologically Sweeping the Visibility Complex of Polygonal Scenes. *Symposium on Computational Geometry*, 1995.
- [SRF87] Sellis, T., Roussopoulos, N., Faloutsos, C. The R+-tree: a Dynamic Index for Multi-Dimensional Objects. *VLDB*, 1987.
- [SS84] Sharir, M., Schorr, A. On Shortest Paths in Polyhedral Spaces. *STOC*, 1984.
- [THH01] Tung, A., Hou, J., Han, J. Spatial Clustering in the Presence of Obstacles. *ICDE*, 2001.
- [W85] Welzl, E. Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time, *Information Processing Letters* 20, 167-171, 1985.
- [Web] <http://www.maproom.psu.edu/dcw>.