International Journal of Geographical Information Science
Vol. 19, No. 10, November 2005, 1091–1111

Research Article

Query processing in spatial databases containing obstacles

JUN ZHANG†, DIMITRIS PAPADIAS*‡, KYRIAKOS MOURATIDIS‡ and ZHU MANLI§

†School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore, 639798

‡Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

§Institute for Infocomm Research, 21 Heng Mui Keng Terrace, Singapore, 119613

(*Received 8 March 2005; in final form 6 June 2005*)

Despite the existence of obstacles in many database applications, traditional spatial query processing assumes that points in space are directly reachable and utilizes the Euclidean distance metric. 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 neighbours, *e*-distance joins, closest pairs and distance semi-joins, assuming that both data objects and obstacles are indexed by R-trees. The effectiveness of the proposed solutions is verified through extensive experiments.

Keywords: Spatial databases; Query processing; Visibility graph

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-neighbour query', consider figure 1, which 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 neighbour 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

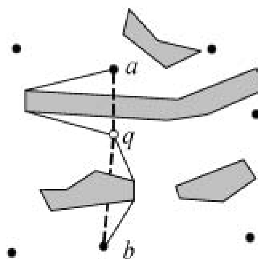


Figure 1. Obstacle nearest-neighbour query example.

*Corresponding author. Email: dimitris@cs.ust.hk

International Journal of Geographical Information Science
ISSN 1365-8816 print/ISSN 1362-3087 online © 2005 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/13658810500286935

same concept applies to any spatial query, e.g. range search, spatial join, and closest pair.

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 figure 1).

In the data-clustering literature, *cod-clarans* (Tung *et al.* 2001) 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 (2001) 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 (Papadias *et al.* 2003), 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 petrol 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. users who need to find the best parking space near their 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 (Guttman 1984, Sellis *et al.* 1987, Becker *et al.* 1990), 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, 6, and 7 describe the algorithms for range search, nearest neighbours, e-distance joins, closest pairs, and distance semi-joins, respectively. Section 8 provides a thorough experimental evaluation, and section 9 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 figure 2, which indexes a set of points $\{a, b, \ldots, k\}$, 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 figure 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.

A *nearest-neighbour* (NN) *query* retrieves the k ($k \geq 1$) data point(s) closest to a query point q. The R-tree NN algorithm proposed by Hjaltason and Samet (1999) 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 figure 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 neighbours (kNN). Furthermore, it is optimal (it visits only the nodes necessary for obtaining the nearest neighbours) and *incremental*, i.e. it reports neighbours in ascending order of their distance to the query point, and can be applied when the number k of nearest neighbours to be retrieved is not known in advance.

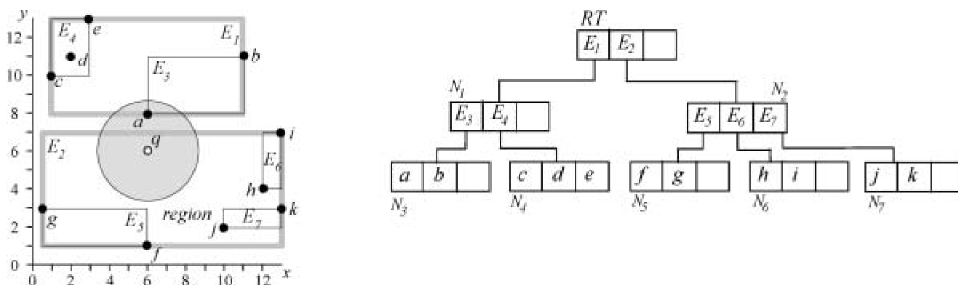


Figure 2. R-tree example.

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 (Brinkhoff *et al.* 1993) synchronously traverses 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 (Corral *et al.* 2000) combine spatial joins with a nearest-neighbour 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-neighbour $t \in T$ (*distance semi-joins* are sometimes referred to as *all nearest-neighbour* queries). This type of query can be answered either (1) by performing a NN query in T for each object in S or (2) by outputting closest pairs incrementally, until the NN for each entity in S is retrieved (Hjaltason and Samet 1999).

2.2 *Query processing in spatial networks*

Papadias *et al.* (2003) study the above query types for spatial network databases, where the network is modelled 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 than 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 neighbours, 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 (de Berg *et al.* 1997). 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. Figure 3(a) shows an example where O contains three obstacles. The corresponding *visibility graph* G is depicted in figure 3(b). 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.

It can be shown (Lozano-Pérez and Wesley 1979) that the shortest path contains only edges of the visibility graph. Therefore, the original problem can be solved by: (1) constructing G and (2) computing the shortest path between p_{start} and p_{end} in G. For the second task, any conventional shortest-path algorithm (Dijkstra 1959, Kung *et al.* 1986) suffices. Therefore, we focus 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 a running time of $O(n^3)$, where n is the number of nodes in G. In order to reduce the cost, Sharir and Schorr (1984) perform a rotational plane-sweep for each graph node and find all the other nodes that are visible to it. Using the example of figure 3, we present the main idea of the algorithm by assuming that node p_{start} is being processed. The target is to find all the nodes that are visible from p_{start}, so that the corresponding edges can be added to G. The algorithm starts with a sweep line (horizontal line in figure 4(a)). The event points for plane-sweep are defined by the nodes of G (excluding p_{start}).

The obstacle edges intersecting the sweep line are maintained in a binary search tree to support visibility checking. For instance, the edges (e_1, e_2, e_3, e_4) intersecting

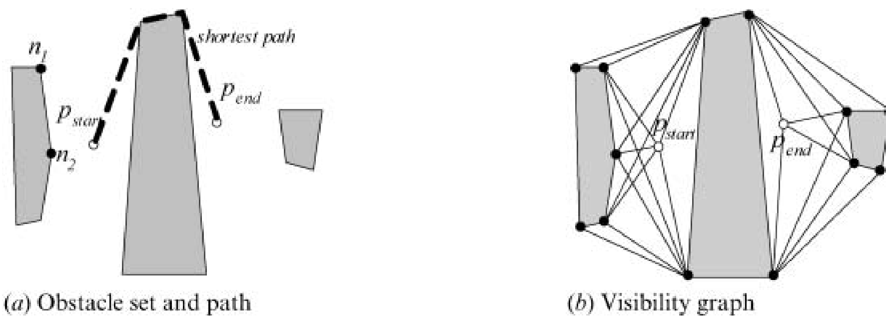


Figure 3. Obstacle path example.

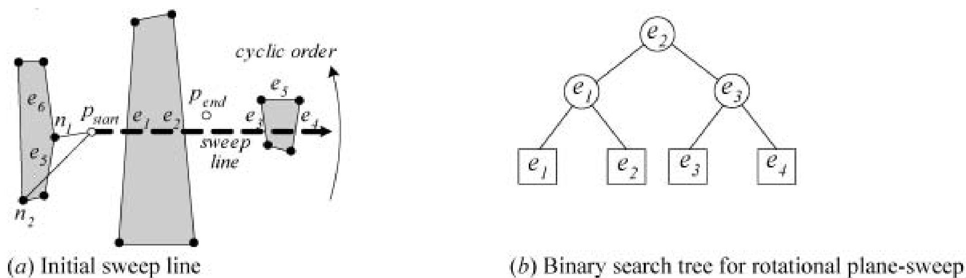


Figure 4. Example of rotational plane sweep.

the initial position of the sweep line are stored in the tree of figure 4(*b*), according to their distance from p_{start}, e.g. those that are farther than e_2 are stored in the right subtree of the root, and those with a distance smaller or equal (to p_{start}) in the left subtree. The nodes are sorted according to their cyclic (counter-clockwise) order around p_{start} and considered in this order. When a node n is encountered, the algorithm performs the following operations: (1) it searches the tree to determine if n is visible from p_{start} (i.e. whether the distance between p_{start} and n is smaller or equal to the distances between p_{start} and all the edges currently in the tree); if yes, it adds an edge $p_{start}n$ to G; (2) it inserts to the tree the incident edges (of n) that lie on the counter-clockwise side of the half-line $p_{start}n$; and (3) it removes from the tree the incident edges that lie on the clockwise side of $p_{start}n$ (such edges cannot obstruct any node that will be considered after n). The sorting and the plane-sweep for each node take a time of $O(n \log n)$. Since the process has to be repeated for all nodes, the total cost is $O(n^2 \log n)$.

Subsequent techniques for visibility graph construction involve sophisticated data structures and algorithms, which are mostly of theoretical interest. The worst case optimal algorithm (Welzl 1985, Asano *et al.* 1986) performs a rotational plane-sweep for all the vertices simultaneously and runs in $O(n^2)$ time. The optimal output-sensitive approaches (Ghosh and Mount 1987, Rivière 1995, Pocchiola and Vegter 1996) have a running time of $O(m+n \log n)$, where m is the number of edges in G. If all obstacles are convex, it is sufficient to consider the *tangent visibility graph* (Pocchiola and Vegter 1995), 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 than 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 online, 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: (1) 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; (2) it finds the set O' of

obstacles that are relevant to the query; (3) it builds a local visibility graph G' containing the elements of P' and O'; (4) 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 figure 5(a), where the shaded areas represent obstacles and points correspond to entities. Clearly, the set P' of entities intersecting the disk C centred 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 figure 5(a)) has a 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 (centred at q with radius e) on the R-tree of O. The local visibility graph G' for the example of figure 5(a) is shown in figure 5(b). For constructing the graph, we use the algorithm of (Sharir and Schorr 1984), 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 that employed by Dijkstra's algorithm (Dijkstra 1959). Specifically, OR maintains a priority queue Q, which initially contains the neighbours of q (i.e. n_1 to n_4 in figure 5(b)) sorted by their obstructed distance. Since these neighbours are directly connected to q, the obstructed distance $d_O(n_i,q)$, $1 \leqslant i \leqslant 4$, equals the Euclidean distance $d_E(n_i,q)$. The first node (n_1) is de-queued and inserted into a set of visited nodes V. For each unvisited neighbour 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. $d_O(n_x,q)=d_O(n_1,q)+d_E(n_x,n_1)$. If $d_O(n_x,q) \leqslant e$, n_x is inserted in Q. Figure 6 illustrates the 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 figure 5(b), 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.

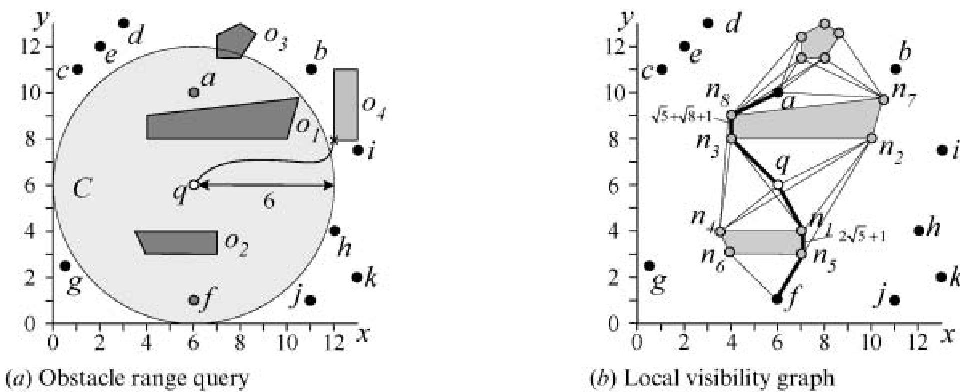


Figure 5. Example of obstacle-range query.

```
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' = Euclidean_range(RT_P, q, e) // get qualifying entities
O' = Euclidean_range(RT_O, q, e) // get relevant obstacles
G' = build_visibility_graph(q, P',O') // algorithm of (Sharir et al.1984)
V = ∅; R = ∅ // V is the set of visited nodes, R is the result
insert <q, 0> into Q
while Q and P' are both non-empty
  de-queue <n, d_O(n,q)> from Q // n has the minimum d_O(n,q)
  if n∉V // n is an unvisited node
    if n∈P' // n is an unreported entity
      R = R ∪ {n}; P' = P' - {n}
    for each neighbour node n_x of n
      if (n_x∉V)
        d_O(n_x,q) = d_O(n,q) + d_E(n,n_x)
        if (d_O(n_x,q) ≤ e)
          insert <n_x, d_O(n_x,q)> into Q
    V = V ∪ n
return R // R is the set of objects within obstructed distance e from q
End OR
```

Figure 6. OR algorithm.

4. Obstacle nearest-neighbour query

Given a query point q, an obstacle set O and an entity set P, an obstacle nearest-neighbour (ONN) query returns the k objects of P that have the smallest obstructed distances from q. Assuming, for simplicity, the retrieval of a single neighbour (k=1) in figure 7, we illustrate the general idea of ONN algorithm before going into detail. First, the Euclidean nearest neighbour of q (object a) is retrieved from P using an incremental algorithm (e.g. Hjaltason and Samet 1999 in section 2.1), and $d_O(a,q)$ is computed. Due to the Euclidean lower-bound property, objects with a potentially smaller obstructed distance than a should be within Euclidean distance $d_{Emax}=d_O(a,q)$. Then, the next Euclidean neighbour (f) within the d_{Emax} 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_{Emax} is updated to $d_O(f,q)$ (i.e. d_{Emax} continuously shrinks).

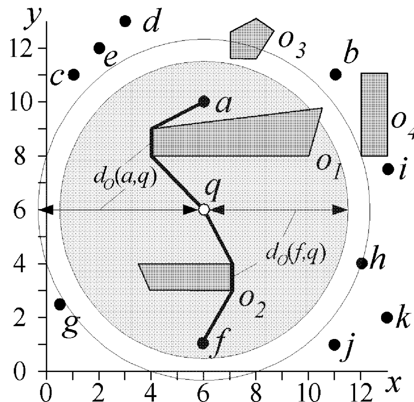


Figure 7. Example of obstacle nearest-neighbour query.

The algorithm terminates when there is no Euclidean nearest neighbour within the d_{Emax} range.

It remains to clarify the obstructed distance computation. Consider, for instance, figure 8 where the Euclidean NN of q is point p. In order to compute $d_{\mathrm{O}}(p,q)$, we first retrieve the obstacles o_1, o_2 within the range $d_{\mathrm{E}}(p,q)$ and build an initial visibility graph that contains o_1, o_2, p, and q. A provisional distance $d_{\mathrm{O1}}(p,q)$ 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 be obstacles (o_3, o_4) outside the range that obstruct the shortest path from q to p.

In order to find such obstacles, we perform a second Euclidean range query on the obstacle R-tree using $d_{\mathrm{O1}}(p,q)$ (i.e. the large circle in figure 8). The new obstacles o_3 and o_4 are added to the visibility graph, and the obstructed distance $d_{\mathrm{O2}}(p,q)$ is computed again. The process has to be repeated, since there may be another obstacle (o_5) outside the range $d_{\mathrm{O2}}(p,q)$ 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 $d_{\mathrm{O}}(p,q)$ (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 figure 9. The initial visibility graph G', passed as a parameter, contains p, q, and the obstacles in the Euclidean range $d_{\mathrm{E}}(p,q)$.

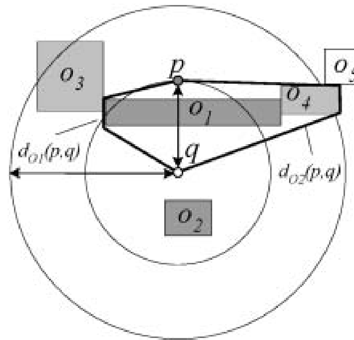


Figure 8. Example of obstructed-distance computation.

Algorithm compute_obstructed_distance(G', p, q)
/* G' is the initial visibility graph, p and q are the two input points */
$d_O(p,q)$= shortest_path_dist(G', p, q)
O' = set of vertices in G'
O_{new}= Euclidean_range(RT_O, q, $d_O(p,q)$)
while ($O' \subset O_{\mathrm{new}}$) // *repeat condition*
  for each obstacle o in $O_{\mathrm{new}} - O'$
    add_obstacle(o,G')
  $d_O(p,q)$= shortest_path_dist(p,q,G')
  $O' = O_{\mathrm{new}}$
  O_{new}= Euclidean_range(RT_O, q, $d_O(p,q)$)
return $d_O(p,q)$ // $d_O(p,q)$ is the obstructed distance between p and q
End compute_obstructed_distance

Figure 9. 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 figure 9 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 figure 8, we want the two nearest neighbours, we reuse 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.

Figure 10 illustrates the complete algorithm for retrieval of k ($\geqslant 1$) nearest neighbours. The k Euclidean NNs are first obtained using the entity R-tree, sorted in ascending order of their obstructed distance to q, and d_{Emax} is set to the distance of the kth point. Similar to the single NN case, the subsequent Euclidean neighbours are retrieved incrementally while maintaining the k (obstructed) NNs and d_{Emax}

```
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 = ∅ // R is the result
P' = Euclidean_NN(RT_P, q, k);
O' = Euclidean_range(RT_O, q, d(p_k ,q)) //p_k is farthest Euclidean NN
G' = build_visibility_graph(q, P',O')
for each entity p_i in P'
  d_O(p_i,q)=compute_obstructed_distance(G',p_i,q) //see Figure 9
  delete_entity(p_i,G')
sort P' in ascending order of d_O(p_i,q) and insert into R
d_Emax= d_O(p_k ,q) // p_k is the farthest NN
repeat
(p,d_E(p,q))=next_Euclidean_NN(RT_P, q);
add_entity(p,G')
d_O(p,q)=compute_obstructed_distance(G',p,q)
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_Emax = d_O(p_k ,q) // update the Euclidean threshold
until d_E(p,q)>d_Emax
return R // R is the set of k-nearest neighbours to q
End ONN
```

Figure 10. ONN algorithm.

(except that d_{Emax} equals the obstructed distance of the kth neighbour), until the next Euclidean NN has a larger Euclidean distance than d_{Emax}.

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 $d_O(s,t) \leqslant e$. Based on the Euclidean lower-bound property, the ODJ algorithm processes an obstacle e-distance join as follows: (1) it performs an Euclidean e-distance join on the R-trees of S and T to retrieve entity pairs (s,t) with $d_E(s,t) \leqslant e$; (2) it evaluates $d_O(s,t)$ for each candidate pair (s,t) and removes false hits. The R-tree join algorithm (Brinkhoff *et al.* 1993) (see section 2.1) is applied for step (1). For step (2), we use the obstructed distance computation algorithm of figure 9.

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 (figure 6). 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 (see Bially 1969). The pseudo-code of the algorithm is shown in figure 11.

6. Obstacle closest-pair query

Given an obstacle set O, two entity datasets S, T, and a value $k \geqslant 1$, an obstacle closest-pair (OCP) query retrieves the k entity pairs (s, t), $s \in S$, $t \in T$, that have the

```
Algorithm ODJ(RT_S, RT_T, RT_O, e)
/* RT_S and RT_T is the entity R-trees, RT_O is the obstacle R-tree, e is the query range */
R = ∅
Rjoin-res = Euclidean_distance_join(RT_S, RT_T, e)
compute Q and P
sort Q according to Hilbert order
for each object q ∈ Q
    P' = set of objects ∈ 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 ∪ {<q, r>/ r ∈ R'}
return R // R is the set of result object pairs
End ODJ
```

Figure 11. ODJ algorithm.

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: (1) performs an incremental closest pair query (Corral *et al.* 2000) on the entity R-trees of S and T, and retrieves the Euclidean closest pair (s,t); (2) evaluates $d_O(s,t)$ and uses it as a bound d_{Emax} for Euclidean closest-pairs search; (3) obtains the next closest pair (within Euclidean distance d_{Emax}), evaluates its obstructed distance and updates the result and d_{Emax} if necessary; (4) repeats step (3) until the incremental search for pairs exceeds d_{Emax}.

Figure 12 shows the OCP algorithm for retrieval of k closest pairs. In particular, OCP first finds the k Euclidean pairs, evaluates their obstructed distances, and treats the maximum distance as d_{Emax}. Subsequent candidate pairs are retrieved incrementally, continuously updating the result and d_{Emax} until no pairs are found within the d_{Emax} 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 pair distances), without a pre-defined termination condition. Another scenario where incremental processing is useful concerns complex queries: 'find the city with more than 1 M 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 *i*OCP (for incremental), shown in figure 13. When an Euclidean CP (s, t) is obtained, its obstructed distance $d_O(s, t)$ is computed, and the entry $<(s, t), d_O(s, t)>$ is inserted into a queue Q. The observation is that all the pairs (s_i, t_j) in Q such that $d_O(s_i, t_j) \leqslant d_E(s, t)$, 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.

7. Obstacle distance semi-join

Given an obstacle set O and two entity datasets S and T, an obstacle distance semi-join (ODS) returns for each point $s \in S$ its obstacle NN $t \in T$. A real-world query of

```
Algorithm OCP(RT_S, RT_T, RT_O, k)
/* RT_S and RT_T is the entity R-trees, RT_O is the obstacle R-tree, k is the number of
pairs requested */
{(s_1, t_1), … , (s_k, t_k)} = Euclidean_CP(RT_S, RT_T, k)
sort (s_i, t_i) in ascending order of their d_O(s_i, t_i)
d_Emax= d_O(s_k, t_k)
repeat
   (s', t') = next_Euclidean_CP(RT_S, RT_T)
   d_O(s', t') = compute_obstructed_distance(G', s', t')
   if (d_O(s', t') < d_Emax)
      delete (s_k, t_k) from {(s_1, t_1), … , (s_k, t_k)} and insert (s', t') in it, so that it remains
sorted by d_O
      d_Emax= d_O(s_k, t_k)
until d_E(s', t') > d_Emax
return {(s_1, t_1), … , (s_k, t_k)}
End OCP
```

Figure 12. OCP algorithm.

```
Algorithm iOCP(RT_S, RT_T, RT_O, TC)
/* RT_S and RT_T is the entity R-trees, RT_O is the obstacle R-tree, TC is the termination
condition */
repeat
    (s, t) = next_Euclidean_CP(RT_S, RT_T)
    d_O(s, t) = compute_obstructed_distance(s, t)
    insert < (s, t), d_O(s, t)> into Q
    for each (s_i, t_j) such that d_O(s_i, t_j) ≤ d_E(s, t)
        de-heap <(s_i, t_j), d_O(s_i, t_j)> from Q
        report <(s_i, t_j), d_O(s_i, t_j)>
until TC
return
End iOCP
```

Figure 13. *i*OCP algorithm.

this form could be: 'for each hotel in a city, find the closest cinema in terms of (obstructed) walking distance'. As discussed in section 2.1, two algorithms can be used to process Euclidean distance semi-joins: the first performs a NN query in T for each object in S, and the second algorithm outputs closest pairs incrementally, until the NN for each entity in S is found. Both approaches can be adapted in our case, by simply replacing the Euclidean with the obstructed distance metric. The problem of the second approach is that subsequent pairs reported by the *i*OCP algorithm do not exhibit locality, and therefore, the local visibility graph for each pair has to be computed from scratch. Furthermore, when the obstructed distance between some points in S and their NN is large (due to different data distributions), a high percentage of the total number of pairs in $S \times T$ must be reported (by *i*OCP) before the query result is complete.

Instead, we adopt the first approach, i.e. we perform $|S|$ ONN queries, but reuse the already computed visibility graphs. In order to achieve locality, we (1) sequentially process all objects of the same leaf node of S sorted by their Hilbert value and (2) consider leaf nodes according to the Hilbert value of their centroids. In this way, entities that are close in space are processed immediately after each other using visibility graphs with many common components. Furthermore, external sorting of S is avoided, since the Hilbert value of leaf nodes can be obtained by traversing the tree, but stopping above the leaf level. The Hilbert values of actual objects are obtained when the corresponding leaf node is loaded.

The next question is how to effectively update the visibility graph and at the same time not exceed the available memory space. When new obstacles or entities have to be included in the visibility graph, we use the *add_obstacle* and *add_entity* functions discussed in section 4. If the updated graph does not fit in memory, we evict a sufficient number of entities (of T) and obstacles, starting with the farthest ones (in terms of obstructed distance) from the current entity $s \in S$ being processed. If these entities or obstacles are needed for future computations, they are reloaded. After the processing of an entity s terminates, s is removed from the graph with all the edges associated with it (we use the function *delete_entity*, as presented in section 4). The algorithm for visibility graph updating (VGU) is shown in figure 14.

The ODS algorithm is a transformed version of the ONN presented in section 4, which, instead of building a new visibility graph, applies VGU to augment the graph of the previously processed point. Owing to the expected high locality of subsequent

```
Algorithm VGU(G', O_new, T_new, s)
/* G' is the current Visibility Graph, O_new and T_new are the obstacle and entity sets to
be included in G' (O_new⊂O, T_new ⊂T ), s is the query point (s ∈ S) */
V = set of vertices in G'
add_entity(s, G');
   for each new obstacle o ∈ O_new – V that must be included in G'
      if there is not enough space in memory
         evict obstacles and/or entities with the highest d_O from s, to free up sufficient
space for the newcomer;
         add_obstacle(o, G');
   for each t ∈ T_new – V
      if there is not enough space in memory
         evict obstacles and/or entities with the highest d_O from s, to free up sufficient
space for newcomer;
      add_entity(t, G');
return G' // G' is the updated Visibility Graph
End VGU
```

Figure 14. VGU algorithm.

entities (achieved by the application of Hilbert order in the R-tree nodes), the graph update operations are minimized.

8. Experiments

In this section, we experimentally evaluate the CPU time and I/O cost of the proposed algorithms, using a Pentium III 733 MHz PC. We employ R*-trees (Becker *et al.* 1990), 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 (see Penn State University Libraries) (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-neighbour algorithms, we execute workloads of 200 queries, which also follow the obstacle distribution.

8.1. *Range queries*

First, we present our experimental results on obstacle range queries. Figure 15(*a*) and (*b*) 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 grow with $|P|$. However, the page accesses for obstacle retrieval remain stable, since the number of obstacles that participate in the distance computations (i.e. those 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).

Figure 16 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

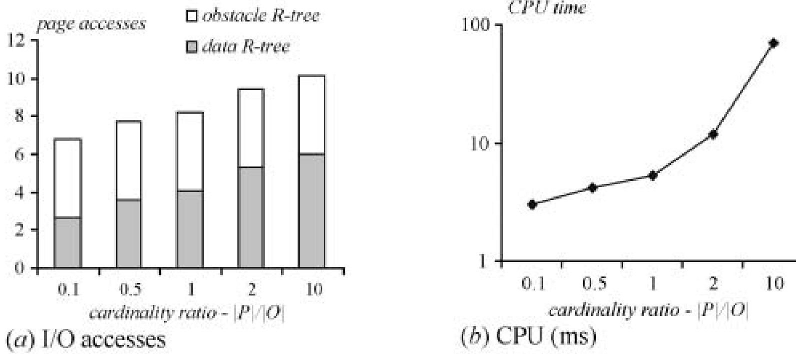


Figure 15. Cost vs |P|/|O| (e=0.1%).

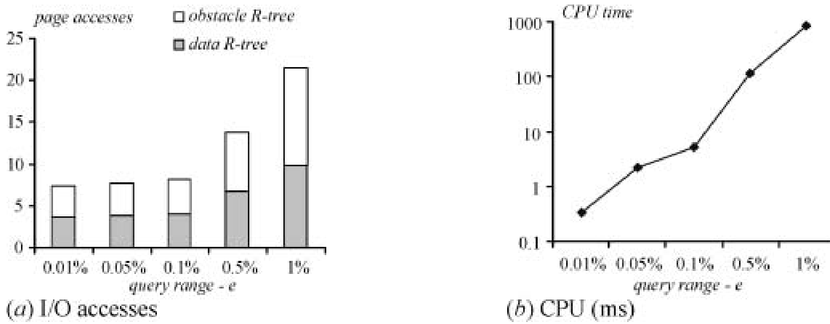


Figure 16. Cost vs *e* (|P|=|O|).

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.

The next experiment evaluates the number of false hits, i.e. objects within the Euclidean, but not in the obstructed range. Figure 17(*a*) 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|). Figure 17(*b*) shows the false-hit ratio

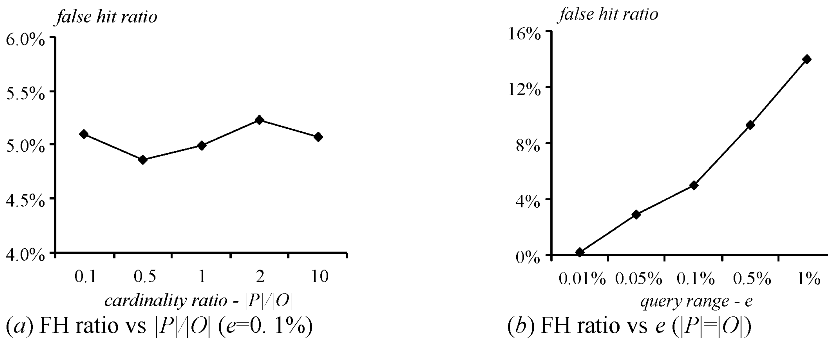


Figure 17. False hit ratio by OR.

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 are 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.

8.2 *Nearest-neighbour queries*

This set of experiments focuses on obstacle nearest-neighbour queries. Figure 18 illustrates the costs of the ONN algorithm as a function of the ratio $|P|/|O|$, fixing the number k of neighbours to 16. The page accesses of the entity R-tree are not seriously affected by $|P|/|O|$ because, as the density increases, the range around the query point where the Euclidean neighbours are found decreases. As a result, the obstacle search radius (and the number of obstacles that participate in the obstructed distance computations) also declines. Figure 18(*b*) confirms this observation, showing that the CPU time drops significantly with data density.

Figure 19 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. Figure 20(*a*) 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 a high false-hit ratio, which

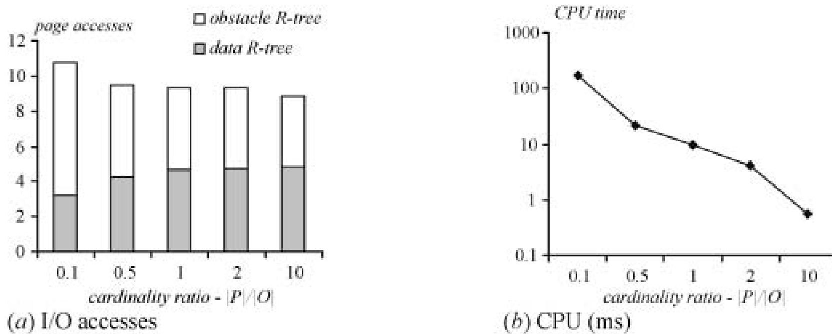


Figure 18. Cost vs $|P|/|O|$ ($k=16$).

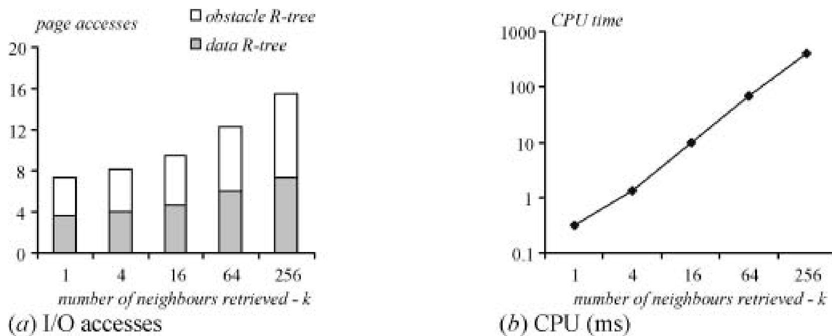
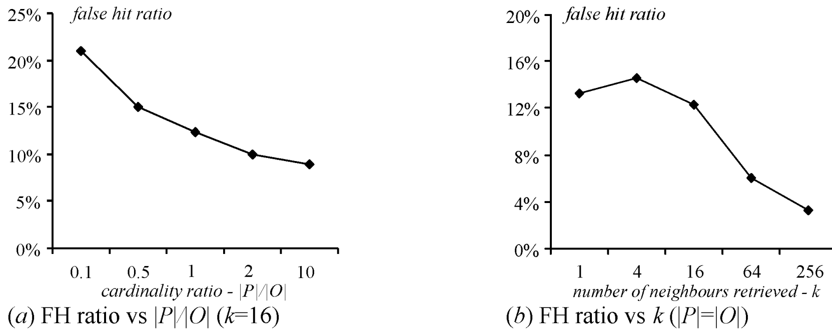


Figure 19. Cost vs k ($|P|=|O|$).



(*a*) FH ratio vs $|P|/|O|$ (k=16) (*b*) FH ratio vs k ($|P|=|O|$)

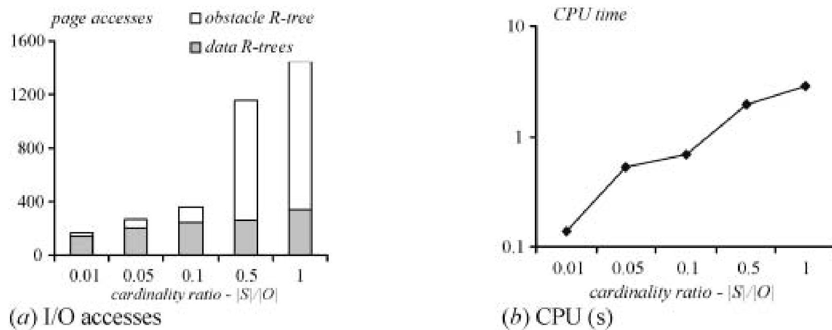
Figure 20. False-hit ratio by ONN.

is gradually alleviated as $|P|$ increases. In Figure 20(*b*), 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 large portion of the k actual (obstructed) NN, despite their probably different internal ordering (e.g. the 1st Euclidean NN is 3rd obstructed NN).

8.3 *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. Figure 21(*a*) 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 more slowly than the obstacle R-tree because the cost of the Euclidean join is not very sensitive to the data density. On the other hand, the output size (of the Euclidean join) grows rapidly 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 figure 21(*b*) which shows the CPU cost as a function of $|S|/|O|$.

In figure 22(*a*), 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 figure 22(*a*), the output of the Euclidean joins (and the number of obstructed distance computations) grows



(*a*) I/O accesses (*b*) CPU (s)

Figure 21. Cost vs $|S|/|O|$ (e=0.01%, $|T|=0.1|O|$).

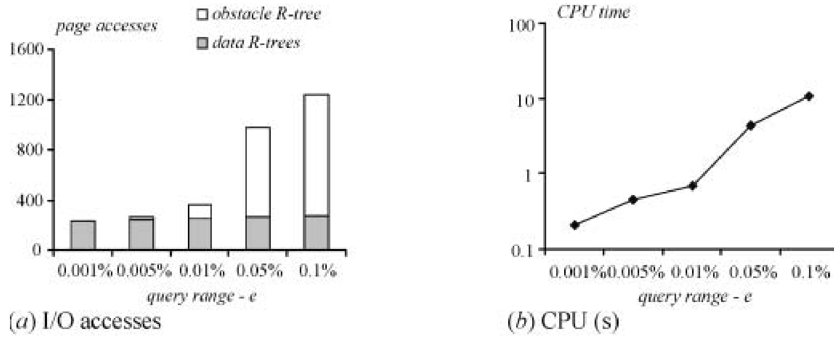


Figure 22. Cost vs e ($|S|=|T|=0.1|O|$).

rapidly with e, which is reflected in the page accesses for the obstacle R-tree and the CPU time (figure 22(b)).

8.4 *Closest pairs*

Next, we evaluate the performance of closest pairs in the presence of obstacles. Figure 23 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 by Corral *et al.* (2000)). On the other hand, the density of S does not significantly affect the accesses to the obstacle R-tree because a high density leads to a closer distance between the Euclidean pairs. The CPU time of the algorithm (shown in figure 23(b)) grows rapidly with $|S|/|O|$, because the dominant factor is the computation required for obtaining the Euclidean closest pairs (as opposed to obstructed distances).

Figure 24 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.

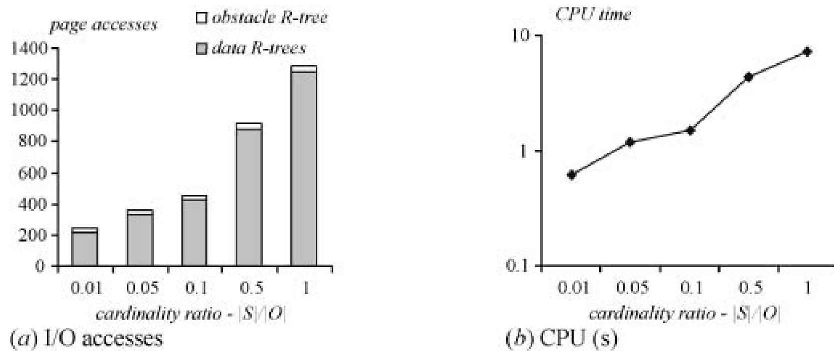
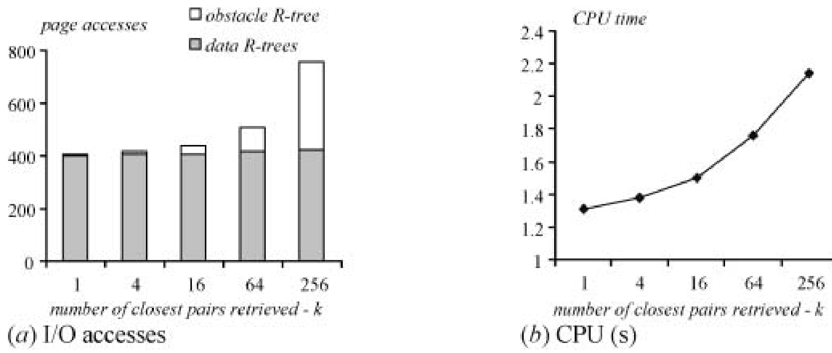


Figure 23. Cost vs $|S|/|O|$ ($k=16$, $|T|=0.1|O|$).



(a) I/O accesses (b) CPU (s)

Figure 24. Cost vs k ($|S|=|T|=0.1|O|$).

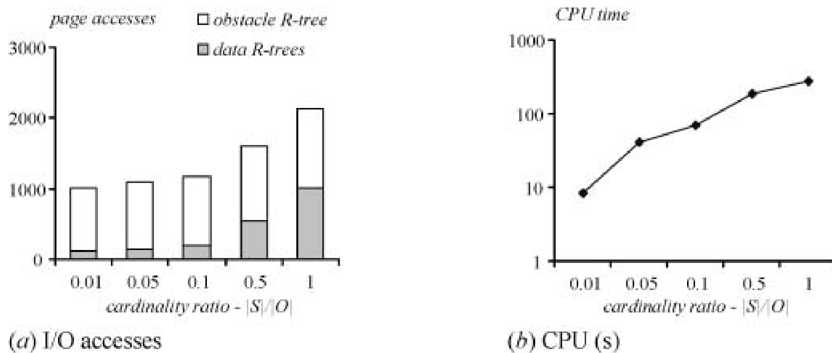
8.5 *Distance semi-joins*

We evaluate the performance of the ODS algorithm for distance semi-joins by varying $|S|$ and $|T|$. Initially, we fix $|T|=0.1|O|$ and plot the number of disk accesses (figure 25(a)) and CPU time (figure 25(b)) as a function of $|S|/|O|$, which ranges from 0.01 to 1. The number of page accesses to the entity R-trees increases with $|S|$, because each object of S necessitates an NN query (thus, S has to be scanned). On the other hand, the I/O cost for the obstacle R-tree is almost stable because for a large $|S|/|O|$, although we need to perform more NN queries, each query (1) reuses the already-computed visibility graph (of the previous query) and (2) requires fewer node accesses to the obstacle R-tree, since the distance where the Euclidean NN is found is smaller (similar to figure 18(a)). The CPU time grows due to the increasing number of queries with $S|/|O|$.

Finally, figure 26 shows the cost of ODS for fixed $|S|=0.1|O|$ and $|T|$ ranging from $0.01|O|$ to $|O|$. The I/O cost for the entity R-trees grows with $|T|$ because each Euclidean NN query becomes more expensive. The page accesses to the obstacle R-tree and the CPU time, however, decrease with $|T|/|O|$, because (1) the number of obstructed distance computations is stable (it only depends on $|S|$), and (2) each computation requires fewer obstacles.

9. Conclusion

This paper tackles spatial query processing in the presence of obstacles. Given one or two entity datasets and a set of polygonal obstacles, our aim is to answer spatial



(a) I/O accesses (b) CPU (s)

Figure 25. Cost vs $|S|/|O|$ ($|T|=0.1|O|$).

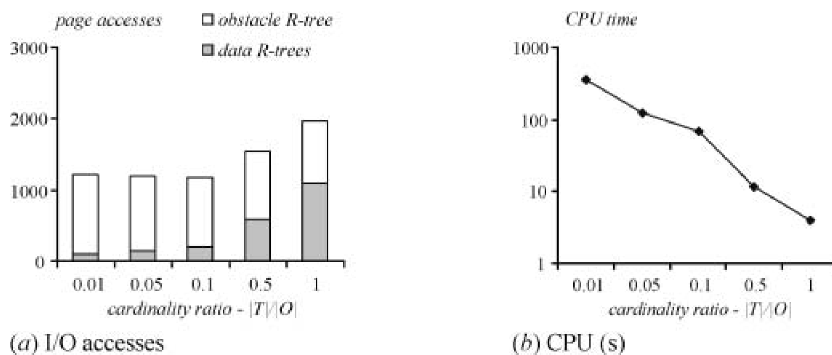


Figure 26. Cost vs $|T|/|O|$ ($|S|=0.1|O|$).

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 the aforementioned fields, we propose an integrated framework that efficiently answers most types of spatial queries (i.e. range search, nearest neighbours, *e*-distance joins, closest pairs, and distance semi joins), subject to obstacle avoidance. Our solutions exploit local visibility graphs and effective R-tree algorithms to achieve efficiency both in terms of *I/O* cost and CPU time. 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.

Acknowledgements
This research was supported by the grant HKUST 6178/04E from Hong Kong RGC and the grant RGg/05 form NTU URC/AcRF.

References

ASANO, T., GUIBAS, L., HERSHBERGER, J. and IMAI, H., 1986, Visibility of disjoint polygons. *Algorithmica*, **1**, pp. 49–63.

BECKER, B., KRIEGEL, H., SCHNEIDER, R. and SEEGER, B., 1990, The R*-tree: an efficient and robust access method. In *9th ACM SIGMOD International Conference on Management of Data*, Atlantic City, NY, 23–25 May (ACM Press), pp. 322–331.

BIALLY, T., 1969, Space-filling curves: their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, **15**, pp. 658–664.

BRINKHOFF, T., KRIEGEL, H. and SEEGER, B., 1993, Efficient processing of spatial joins using R-trees. In *12th ACM SIGMOD International Conference on Management of Data*, Washington, DC, 26–28 May (ACM Press), pp. 237–246.

CORRAL, A., MANOLOPOULOS, Y., THEODORIDIS, Y. and VASSILAKOPOULOS, M., 2000, Closest pair queries in spatial databases. In *9th ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 16–18 May (ACM Press), pp. 189–200.

DE BERG, M., VAN KREVELD, M., OVERMARS, M. and SCHWARZKOPF, O., 1997, *Computational Geometry*, pp. 305–315 (Berlin: Springer).

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

ESTIVILL-CASTRO, V. and LEE, I., 2001, Fast spatial clustering with different metrics in the presence of obstacles. In *9th ACM International Symposium on Advances in Geographic Information Systems*, Atlanta, GA, 9–10 November (ACM Press), pp. 142–147.

GHOSH, S. and MOUNT, D., 1987, An output sensitive algorithm for computing visibility graphs. *IEEE Foundations of Computer Science*.

GUTTMAN, A., 1984, R-trees: A dynamic index structure for spatial searching. In *3rd ACM SIGMOD International Conference on Management of Data*, Boston, MA, 18–21 June (ACM Press), pp. 47–57.

HJALTASON, G. and SAMET, H., 1999, Distance browsing in spatial databases. *ACM Transactions on Database Systems*, **24**, pp. 265–318.

KUNG, R., HANSON, E., IOANNIDIS, Y., SELLIS, T., SHAPIRO, L. and STONEBRAKER, M., 1986, Heuristic search in data base systems. In *1st International Conference on Expert Database Systems*, Kiawah island, SC (Benjamin-Cummings Publishing), pp. 537–548.

LOZANO-PÉREZ, T. and WESLEY, M., 1979, An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of ACM*, **22**, pp. 560–570.

PAPADIAS, D., ZHANG, J., MAMOULIS, N. and TAO, Y., 2003, Query processing in spatial network databases. In *29th International Conference on Very Large Data Bases*, Berlin, Germany, 9–12 September (Morgan Kaufmann), pp. 790–801.

POCCHIOLA, M. and VEGTER, G., 1995, Minimal tangent visibility graph. *Computational Geometry: Theory and Applications*, **6**, pp. 303–314.

POCCHIOLA, M. and VEGTER, G., 1996, Topologically sweeping visibility complexes via pseudo-triangulations. *Discrete Computational Geometry*, **16**, pp. 419–453.

RIVIÈRE, S., 1995, Topologically sweeping the visibility complex of polygonal scenes. In *11th Annual ACM Symposium on Computational Geometry*, Vancouver, British Columsia, 5–7 June (ACM Press), pp. 436–437.

SELLIS, T., ROUSSOPOULOS, N. and FALOUTSOS, C., 1987, The R+-tree: a dynamic index for multi-dimensional objects. In *13th International Conference on Very Large Data Bases*, Brighton, UK, 1–4 September (Morgan Kaufmann), pp. 507–518.

SHARIR, M. and SCHORR, A., 1984, On shortest paths in polyhedral spaces. In *16th Annual ACM Symposium on Theory of Computing*.

TUNG, A., HOU, J. and HAN, J., 2001, Spatial clustering in the presence of obstacles. In *17th International Conference on Data Engineering*, Heidelberg, Germany, 2–6 April (IEEE Computer Society), pp. 359–367.

WELZL, E., 1985, Constructing the visibility graph for n line segments in $O(n^2)$ time. *Information Processing Letter*, **20**, pp. 167–171.

Penn State University Libraries, Available online at: http://www.maproom.psu.edu/dcw (accessed 14 October 2005).