Knowledge and Information Systems manuscript No.
(will be inserted by the editor)

The largest empty circle with location constraints in spatial databases

Gilberto Gutiérrez · Juan R. López · José R. Paramá · Miguel R. Penabad

Received: date / Accepted: date

Abstract Given a set S of points in the two-dimensional space, which are stored in a spatial database, this paper presents an efficient algorithm to find the empty circle, in the area delimited by those points, with the largest area and containing only a query point q.

Our algorithm adapts previous work in the field of computational geometry to be used in spatial databases, which require to manage large amounts of data. To achieve this objective, the basic idea is to discard a large part of the points of S, in such a way that the problem can be solved providing only the remaining points to a classical computational geometry algorithm that, by processing a smaller collection of points, saves main memory resources and computation time.

The correctness of our algorithm is formally proven. In addition, we empirically show its efficiency and scalability by running a set of experiments using both synthetic and real data.

Keywords Spatial databases · query processing · geographical information systems · largest empty circle.

1 Introduction

Spatial databases (SDBs) are a core component of geographical information systems (GISs). With respect to classical databases, the geographical component re-

For the first author, this work was supported by the research projects DIUBB [140515 3/R; 142719 3/R]s and MECESUP [UBB0704]. For the rest of authors by Ministerio de Economía y Competitividad (PGE and FEDER) under grants [TIN2016-78011-C4-1-R, TIN2013-46238-C4-3-R] and Centro para el desarrollo Tecnológico e Industrial under grants [IDI-20141259, ITC-20151305, ITC-20151247]. The authors also want to thank Boaz Ben Moshe for sharing his Java Delaunay code and for kindly answering our questions.

Juan R. López · José R. Paramá · Miguel R. Penabad
Universidade da Coruña, Facultade de Informática, CITIC, Campus de Elviña, 15071 A Coruña, Spain E-mail: {juan.ramon.lopez, jose.parama, miguel.penabad}@udc.es

Gilberto Gutiérrez
Universidad del Bío-Bío, Computer Science and Information Technologies Department, Chillán, Chile E-mail: ggutierr@ubiobio.cl

quires the design of new data structures, spatial access methods, query languages, and algorithms to manage large amounts of this type of information. Computational geometry is an important source of knowledge and algorithms, thus many query types included in SDBs are problems that were first tackled in this field. However, the inclusion of these new queries in SDBs requires further developments, mainly due to the management of huge amounts of data.

Finding large empty areas in a space that contains a set of points is a well known problem in computational geometry. The target is to find the largest geometrical figure that contains no points [45,34,1,36,12,4,2,31,33,18].

This work deals with the Largest Empty Circle (LEC) problem in a space containing a set S of points, which obtains the LEC having its center in the convex hull of S. The convex hull of a set of points in the two-dimensional space is the smallest convex polygon that contains S. The LEC problem is sometimes referred to as the Toxic Waste Dump Problem or Obnoxious Facility Location [4]: given a set of cities, the less hazardous place to dump a toxic waste would be the place that maximizes the distance to any of those cities. That place would be the center of the LEC [45].

The LEC problem, as most of the computational geometry problems described above, has two variants: (a) find the LEC with no constraints, and (b) find the LEC that contains a point q that does not belong to the set S. This point is known either as the query point or as a location constraint for the LEC. Figure 1 shows an example of the two versions of the problem.

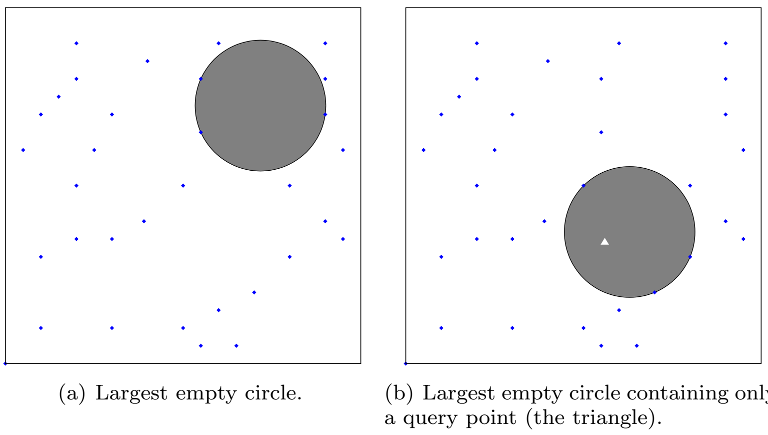

(a) Largest empty circle. (b) Largest empty circle containing only a query point (the triangle).

Fig. 1 Two variants of the LEC problem.

In this paper, we deal with the second version of the problem, because the inclusion of this constraint can be of interest to solve some kinds of problems. For example, consider a scientist who desires to find a location for a prospection that requires the use of explosives. Assume that she has access to a system that stores the locations of villages, isolated houses, etc., as points. Note that if she asks for the largest empty circle, the answer could be in an area not suitable, because, for example, the soil does not have the adequate characteristics, the land is protected, etc. Instead, it is likely that she has a list of candidate placements for her prospection. Therefore, the best option is to mark a point at each candidate placement, and issue a LEC query at each of those points. From all the resulting empty circles, she will be able to chose the best option (probably the one with the biggest area) with the goal of disturbing the fewest possible people.

Apart from GIS and SDBs, this problem is also of interest in the field of sensor networks [3], specifically to design algorithms to deploy sensors [14, 20, 47], to solve coverage problems [13], or to design routing protocols [44]. Other fields where the LEC problem has applications are very-large scale integration circuits [30] or data mining [19].

Very efficient algorithms have been developed in the area of computational geometry to solve the LEC problem. However, those solutions always assume that the data fits in main memory, something that in SDBs (and thus GIS) is not usually feasible. Moreover, focusing on the case of the LEC containing only a query point, the computational geometry approach is even more inadequate for SDBs. The algorithms in [31, 3, 29] require a preprocessing phase which builds a data structure that, once built, is capable of fastly answering queries (in most cases in $O(\log n)$ time) for any given query point. That preprocessing is costly and requires to keep the resulting data structure in memory in order to be able to answer queries. The algorithm of Kaplan and Sharir [31] requires $O(n \log^2 n)$ time to build the data structure, which needs $O(n \log n)$ space, Agustine et. al. [3] propose two algorithms, with $O(n^{3/2} \log^2 n)$ and $O(n^{5/3} \log n)$ time complexity, and $O(n^{3/2} \log n)$ and $O(n^{5/3})$ space, respectively, and finally, Kaminker and Sharir [29] show an algorithm that requires $O(n \log^3 n)$ time and $O(n)$ space. However, the main problem for SDBs is that the preprocessing builds a data structure for a given dataset, and after changes in the set of points (due to insertions or deletions), that data structure is not valid any longer. Due to the costs of the preprocessing phase, it is not feasible to rebuild the data structure after the insertion or deletion of a point in the database.

Instead, we present here an approach that relies on typical spatial indexes designed for databases, which have two important features: they adapt their structure to updates, insertions, and deletions with little effort, and they can be used without loading all the data structure in main memory. Our algorithm selects a reduced set of points to solve the problem, with the help of a database spatial index, in order to speed up the process. We will prove that applying a computational geometry algorithm to that reduced set of points gives always the correct answer.

As explained, the idea of adapting queries from the computational geometry field to SDBs is not new. Following this approach, several queries have been proposed in the field of SDBs taking advantage of the presence of a multidimensional structure [40, 10, 24, 25, 28, 16, 17, 26]. This paper presents a similar work that deals with the largest empty circle containing a query point q, assuming that the points are indexed by a (hierarchical, dynamic, and multidimensional) spatial index. More specifically, we present an algorithm that makes use of such a spatial index combined with a computational geometry algorithm to obtain the LEC containing a query point. We have implemented our algorithm using two spatial indexes: an R*-tree [8], and a K-D-B-tree [39].

In our experimental section, we compare the efficiency of our algorithm against a classical setup, which consists in loading all points in main memory and then use a classical computational geometry algorithm. The results show that our algorithm is 2.5-981 times faster and require only between 3.71% and 45.03% of the main memory space required by the computational geometry algorithm.

The outline of the paper is as follows: Section 2 introduces some related work. Section 3 presents some background and definitions. Section 4 discusses the basic method to discard points for the computation of the LEC. Section 5 presents our algorithm. Section 6 shows the results of our experiments. Finally, Section 7 discusses our conclusions and directions for future work.

2 Related work

Obtaining the largest empty geometric figure in a space that contains a set of points has been an active research field in the last decades. Shamos and Hoey [43] outlined an algorithm to compute the LEC without location constraints that runs in $O(n \log n)$, being n the number of input points. Toussaint showed that the algorithm of Shamos and Hoey does not work properly in some cases and provided a new algorithm keeping the $O(n \log n)$ time complexity. Later, in [37], some optimizations were provided, but keeping $O(n \log n)$ complexity. There are also works in 3D. In this case the algorithms compute the largest empty sphere [33].

The constrained variant is also an old problem. Moreover, there are several versions. The largest empty circle only containing a query point is tackled in [31, 3, 29], whereas in [5], the LEC is constrained to have only a query line in its interior. Algorithms to compute the LEC where its center is constrained to be within a simple polygon [11, 45, 15] and an arbitrary complex $n - gon$ [45, 15] were also studied.

In the field of SDBs, there are several works that improve a computational geometry algorithm by means of a spatial index (the following examples use an R-tree): in [40], it is presented an algorithm that finds the nearest neighbor to a given point; in [10], it is shown a method to find the convex hull of a set of points stored in a spatial database; in [28, 16, 17, 26], several algorithms to solve the k-pairs ($k \geq 1$) of nearest neighbors between two sets are presented.

To the best of our knowledge, the LEC problem has not been faced in the field of SDBs. However, a similar problem, the largest empty rectangle only containing a query point, was tackled in [25]. Here we apply the same approach described in that work. There, a spatial index is used to obtain some real points that form a barrier that the largest empty rectangle cannot cross, and this barrier is used to reduce the number of input points to a computational geometry algorithm that computes the largest empty rectangle. However, the barrier used for the rectangle problem is not valid at all for the case of the circle, and thus we present a method to delimit the area where the points can be discarded in the case of circles.

3 Background and basics

3.1 Basic definitions

Next, we introduce some definitions that will be used later.

Definition 1 Let S be a set of points in $\Re^2$ and a query point $q \notin S$.

- An *empty circle* does not contain any point of S in its interior.
- A *maximal empty circle* (MEC) is an empty circle such that it is not fully contained in any other empty circle.
- Among the $MECs$, that with the largest radius is the *largest empty circle* (LEC).
- A *maximal empty circle only containing a query point* ($QMEC$) is a MEC only containing the query point q.
- The *largest empty circle only containing a query point* ($QLEC$) is the $QMEC$ with the largest radius.

For a given set of points, Figure 1(a) shows the LEC, and Figure 1(b) shows the QLEC for the same set of points and a query point q.

Definition 2 We say that a circle is *supported* by a point p, if its boundary contains p.

Definition 3 Let p_i, p_j, and p_k three points, we denote as C_{ijk} the unique circle supported by those points.

Definition 4 Given a rectangle $R \subseteq \Re^2$ and a point $p \subset R$, p divides R in four quadrants (see Figure 2(a)):

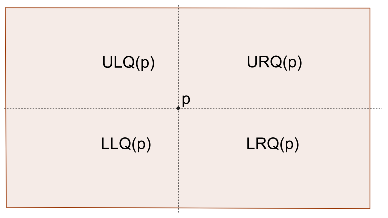
- $ULQ(p)$: the Upper-Left *quadrant* of p is the rectangle bounded by point p and the Upper-Left corner of R.
- $URQ(p)$: the Upper-Right *quadrant* of p is the rectangle bounded by point p and the Upper-Right corner of R.
- $LLQ(p)$: the Lower-Left *quadrant* of p is the rectangle bounded by point p and the Lower-Left corner of R.
- $LRQ(p)$: the Lower-Right *quadrant* of p is the rectangle bounded by point p and the Lower-Right corner of R.

Definition 5 Let P be a convex polygon, and let E_{ij} be an edge of P, defined by two vertices p_i and p_j. Let q be an external point, such that $q \cap P = \emptyset$. We say that the edge E_{ij} of P is *visible* from q if none of the two segments defined by the pairs of points (q, p_i) and (q, p_j) do intersect P, except (precisely) in p_i or p_j, respectively. Otherwise, it is *non-visible*. With $V_q(P)$ we will refer to the set of visible edges of the polygon P from the point q.

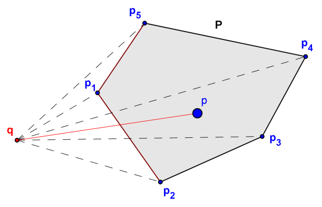
An alternative way to check for the visibility of an edge E_{ij} of P is the following: E_{ij} is visible from q if the line defined by the pair of points (p_i, p_j) completely separates P and q in different half-planes.

Figure 2(b) shows a polygon P and a point q. Only the edges of P defined by the vertex pairs (p_1, p_2) and (p_1, p_5) are visible from q. All the other edges (p_2, p_3), (p_3, p_4) and (p_4, p_5), are non-visible from q. Observe that no point p of P –including those from the visible edges defined by (p_1, p_2) and (p_1, p_5)– can be connected with q by a line or segment without intersecting at least one of the edges in $V_q(P)$.

Our *visibility* definition is based in the concept of *point visibility polygon* from the computational geometry field [35], which refers to the portion of a polygon P visible from the point q, with q inside P (that is, $q \cap P \neq \emptyset$). Some authors define also the term *external visibility* [46], which refers to the portion of P visible from a point q in the complement of P (that is, $q \cap P = \emptyset$, which is the scenario of our definition).



(a) Quadrants defined by a point p.



(b) Visible and non-visible edges of a convex polygon P from a point q.

Fig. 2 Support examples for our definitions.

3.2 Hierarchical and multidimensional spatial indexes

SDBs pose several restrictions over any multidimensional data structure used to index the data.

– It should be a dynamic structure, allowing efficient insertions and deletions of keys, in addition to queries.
– The index is expected to be very large, so it will be necessary to store most of the index on secondary storage.

Many of the spatial indexes that fulfill these requirements are hierarchical tree structures, where all elements in a subtree are bounded by the k-dimensional coordinates of its root key. However, conventional hierarchical tree structures, such as quadtrees [21], kd-trees [9] and many of their variants, do not work well if they are not completely loaded into main memory. As a response, a family of trees where the nodes are adapted to the size of a disk page and with a large fanout appeared, frequently denoted as *tree directory* [41]. Examples are linear quadtrees [22], K-D-B-trees [39], or R-trees [27].

A complete survey of index structures for spatial databases can be found in [6]. This includes data-driven structures, which hierarchically cluster sets of spatial objects, and space-driven structures, where the space is partitioned into rectangular cells that contain the objects. R-trees and their variants are examples of the first type, while grid files, quadtrees, and K-D-B-trees are examples of the second one. We have used in our experiments an R*-tree (a variant of the R-tree) and a K-D-B-tree, so we will briefly describe both data structures.

The R-tree [27] is a balanced tree that stores k-dimensional geometric objects. In inner levels, the objects in the subtrees rooted at a node are represented by the Minimum Bounding k-dimensional Rectangle (MBR) enclosing them. In this paper, we focus on 2 dimensions, therefore these MBRs are rectangles with faces parallel to the coordinate axes.

All leaves are on the same level, and contain (or have pointers to) all the indexed real points (or objects) of the database. Inner nodes are composed by entries of the form $\langle MBR, ref\rangle$, where ref is a pointer to the child node of the entry; and MBR is the MBR that spatially encloses the MBRs (or points) in the subtree rooted at that entry.

Figure 3 shows a space with points, the MBRs enclosing them, and the shape of an R-tree. Dotted lines denote the MBRs of the entries at the root node. The rectangles with solid lines are the MBRs enclosing the points of the leaves.

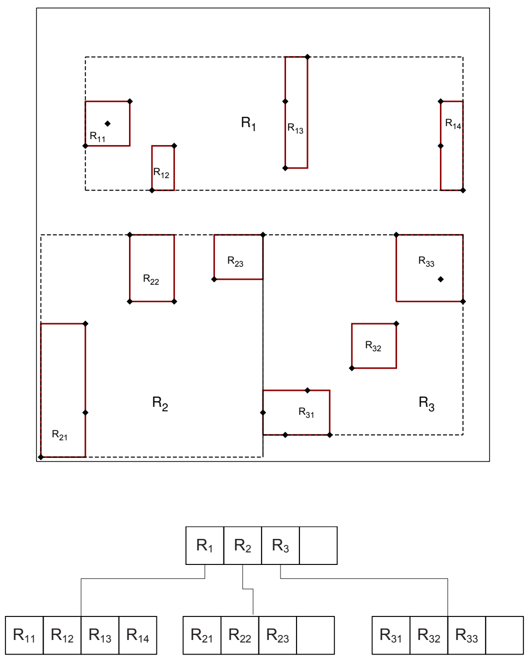


Fig. 3 An R-tree.

The original R-tree designed by Guttman [27] follows one heuristic optimization: to minimize the area of the MBRs of the inner nodes, while keeping the CPU cost low. It usually has as a beneficial side effect of minimizing the overlapping area among the MBRs of inner nodes, although with some geographical distribution of points it may not be the case. Additionally, R-trees do not consider other possible optimizations, such as minimizing the margin (the sum of the lengths of the MBR edges), or storage needs.

Guttman already discussed some variations of the R-tree, mainly trying to improve the part of the algorithm that splits a node (when inserting a new key into an already full node). Other variations of R-trees have appeared in the literature. An exhaustive survey of all variations can be found in [32].

One of the most popular and efficient variations is the R*-tree [8], which tries to improve the performance of the R-tree in two cases: when a new key is inserted in a non full leaf node (only the MBR of the chosen node must be recomputed), and when the insertion is performed on an already full node, and it has to be split.

For the first case, to choose the insertion path for a new element, R-trees usually take into account only the area parameter (how much the area of the

MBR is increased). The R*-tree also takes into account other parameters, like the margin (defined as length of the MBR perimeter) and overlapping of MBRs, trying to minimize all of them. The same parameters (area, margin and overlap) are also considered for the second case, when splitting a node.

With these optimizations, the R*-tree is commonly accepted as one of the most efficient versions of the R-tree. For this reason, we have chosen the R*-tree to perform our experiments.

The K-D-B-tree [39] is another k-dimensional, hierarchical, dynamic index, which tries to combine the benefits of kd-trees and B-trees [7]. kd-trees are binary trees and, as such, they are dynamic (allowing insertions and deletions on logarithmic time), but they were not designed to work on secondary storage. B-trees, on the other hand, are well known for their good performance on secondary storage.

The K-D-B-tree is somehow similar to the R-tree. It is a balanced tree, having all its leaves at the same level. It has two types of nodes or pages: *Point* pages (leaves) and *Region* pages (internal nodes). Each entry (key) in a point page stores a point and the object it identifies, or a reference to it. Each entry in a region page stores a region (a concept analogous to the R-tree MBR) and a page id, which is a pointer to a child node.

The main difference with the R-tree is how the regions are formed. Basically, it starts with a point page (we can consider an initial "virtual" region that is the full k-dimensional space) and, as points are added, it becomes full and must be split into two regions along one of the dimensions. Considering again only 2 dimensions, each region is split either horizontally or vertically into two regions, which can in turn be split again. All dimensions are usually used cyclically, so, if one region comes from a vertical split, this region will be split horizontally, and so on. Regions are always disjoint, and the union of all the regions in a region page is also a region. Figure 4 shows a conceptual example of a 2-dimensional K-D-B-tree. The grayed areas represent the regions that are not part of the region page. Note that points are only present on the leaves or point pages.

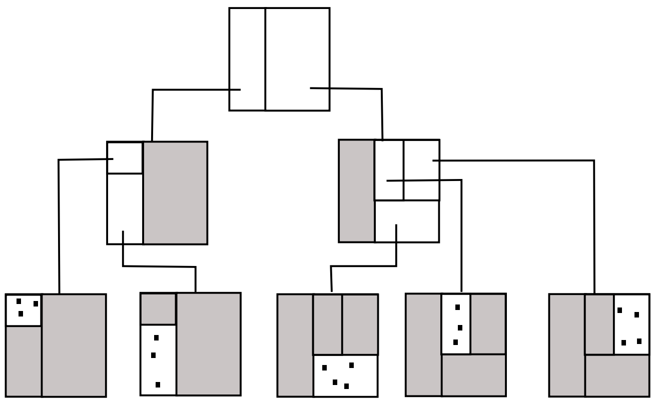


Fig. 4 A K-D-B-tree.

3.3 Delaunay triangulation and convex hull

The following concepts are used later in our algorithm.

Definition 6 Let S be a set of points in $\Re^2$. The Delaunay triangulation of S (see Figure 5(a)), denoted as $Dn(S)$, is a subdivision of $\Re^2$ into triangles such that any two triangles intersect in a common face or not at all, and such that no point in S is inside any circle passing through the three points defining a triangle.

Definition 7 Let S be a set of points in $\Re^2$. The convex hull of S, denoted as $H(S)$, is the smallest convex polygon that contains S (see Figure 5(b)).

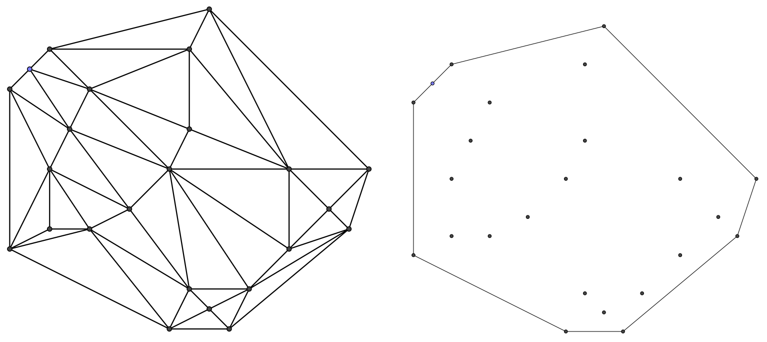

(a) Delaunay triangulation. (b) Convex Hull.

Fig. 5 Different elements computed from a set of points.

Lemma 1 *Let S a set of points in $\Re^2$. Let $Dn(S)$ the Delaunay triangulation of S and $H(S)$ the convex hull of S. The LEC must be supported by the three points defining a triangle of $Dn(S)$, or only by two, but in this case its center must lie on an edge of $H(S)$.*

Proof: By Theorem 6.25 of [42] and given that the Delaunay triangulation of a set of points corresponds to the dual graph of the Voronoi diagram. □

As a reminder, Table 1 shows a brief description of some of definitions introduced in this section.

4 Filtering points

The idea behind our algorithm is to delimit an area around the query point such that the points in that area are enough to obtain the QLEC. The aim is to provide those points as input to a classic computational geometry algorithm to compute the QLEC, whereas the rest are discarded.

The question is how to obtain an area that ensures that the computational geometry algorithm obtains the same QLEC as if we provide the whole set of points.

$QLEC$	The largest maximal empty circle containing the query point
Supported	A circle is supported by a point p, if its boundary contains p.
C_{ijk}	The unique circle supported by the three points p_i, p_j, and p_k
$ULQ(p)$	The Upper-Left *quadrant* of p is the rectangle bounded by point p and the Upper-Left corner of a rectangle R
$URQ(p)$	The Upper-Right *quadrant* of p is the rectangle bounded by point p and the Upper-Right corner of R
$LLQ(p)$	The Lower-Left *quadrant* of p is the rectangle bounded by point p and the Lower-Left corner of R
$LRQ(p)$	The Lower-Right *quadrant* of p is the rectangle bounded by point p and the Lower-Right corner of R
$V_q(P)$	The set of visible edges of a convex polygon P from an exterior point q. An edge E_{ij} of P is visible from q if the line defined by its vertices completely separates q from P in different half-planes
$Dn(S)$	The Delaunay triangulation of a set of points S
$H(S)$	The convex hull of a set of points S

Table 1 Definitions.

Obviously, we want to keep that area as small as possible, since the smaller the area, the fewer points will be provided to the computational geometry algorithm, and thus better running times and less memory consumption will be obtained.

Our method is based on identifying three points defining a triangle that encloses the query point. Those points represent a barrier that any empty circle enclosing the query point cannot cross (although the QLEC is not necessarily supported by them). Next, we define three circles, each one passing through the query point and two of the three surrounding points previously identified (see Figure 6). To compute the solution, the points inside those circles are provided as input to a classic computational algorithm, together with the points defining the convex hull, that that might be necessary to adjust the solution (Lemma 1). All the other points are not necessary and can be discarded.

Lemma 2 *Let S be a set of points in a fixed axis-parallel rectangle $R \subseteq \Re^2$, and let q be a query point $q \notin S$ such that $q \cap R \neq \emptyset$. Let p_i, p_j, and p_k be three points in S that define a triangle T_{ijk} enclosing q. Let S_q be the set of points composed by: p_i, p_j, and p_k, the points in C_{qij}, C_{qjk}, and C_{qik}, and the vertices of $H(S)$. The QLEC obtained from S_q is the same as the one computed from S using q as query point.*

Proof. The target of the proof is to guarantee that S_q contains all the points of S (the original dataset) needed to obtain the correct answer. By Lemma 1, we must consider two possible situations:

– The solution (QLEC) is supported by three points from S and the center of the QLEC lies inside $H(S)$.
– The solution (QLEC) is only supported by two points from S and the center of the QLEC lies on an edge of $H(S)$.

1. First, we consider the case where the QLEC is supported by three points: p_l, p_u and p_v. Since q is inside T_{ijk}, two possibilities arise:
   – The three points p_l, p_u, and p_v are inside T_{ijk}.
   – One or more of the points p_l, p_u, and p_v are outside T_{ijk}.

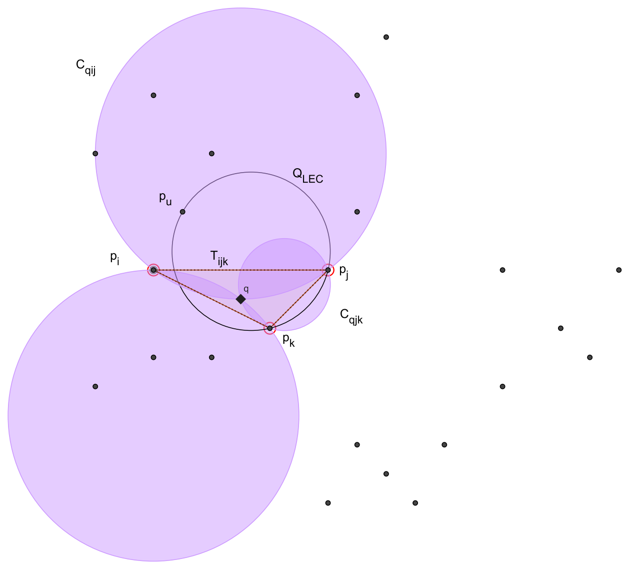


Fig. 6 T_{ijk} and C_{qij}, C_{qjk}, and C_{qjk}

1.1. In the first case, observe that, by construction, S_q has all the points that are within T_{ijk} (see Figure 7(a)), and thus it is enough to compute the QLEC.

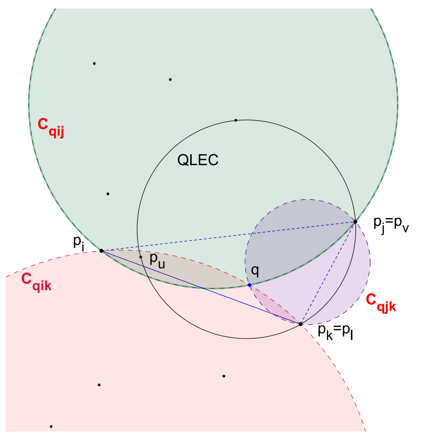
1.2. In the second case, as q is inside T_{ijk}, any circle supported by a point outside T_{ijk} must intersect it to reach q. Let us assume, without loss of generality, that p_u is outside T_{ijk}, and that the edge of T_{ijk} connecting p_i and p_k (denoted as E_{ik}) is the one intersected by a line connecting p_u and q (Figure 7(b)).

Observe that, in that scenario, the QLEC should intersect or completely contain $E_{i\,k}$ to reach q, keeping always p_i and p_k as a barrier for its growth. Indeed, in the best circumstances for the QLEC's size (assuming that no other point represents a barrier), it will be supported both by p_i and p_k. This can be deduced from the fact that any circle supported by p_u, and containing q, may be expanded by moving its center and/or making its radius larger, until reaching both p_i and p_k (Figure 7(b) again).

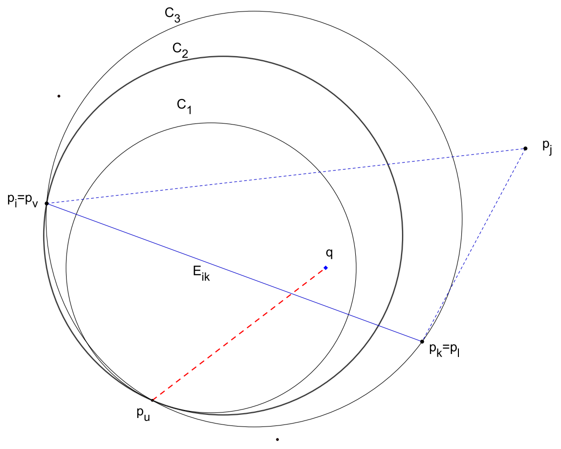
We are going to prove by contradiction that S_q is enough to compute the QLEC. Let us suppose that S_q does not contain the point p_u, which supports the QLEC. Then, we know that:

i. C_{qik} is supported by p_i, p_k, and q.
ii. Assumption: p_u, the point supporting the QLEC, is outside C_{qik} (or it would be included in S_q).

Observe that (Figure 8), from the infinite set of circles supported by p_i and p_k and containing q, C_{qik} is the one that covers, with respect to q, the farthest area towards p_u (that is, in the direction of the face of E_{ik} outside T_{ijk}). Moreover, the farther the circle reaches, the closer to the boundary becomes q. In fact, q lies on the boundary of (supports) C_{qik}.



(a) The case when all the three points (p_u, p_l, and p_v) are inside T_{ijk}.



(b) The case when one point (p_u) is outside T_{ijk}.

Fig. 7 The two main cases of the proof.

We have assumed that the QLEC is supported by p_u. Now, the three following scenarios are possible:

1.2.1 The QLEC is supported also by both p_i and p_k (Figure 8).
To reach p_u, the QLEC should have a bigger area than C_{qik}. However, such a big circle would not contain q. Observe that, if we take C_{qik} and try to transform it into the QLEC, we have to move its center towards p_u to make it bigger. This displacement will move the boundary of C_{qik} in such a way that q will remain outside that circle (see the dotted circle in Figure 8). Therefore, we reach a contradiction.

1.2.2 The QLEC is supported by either p_i or p_k and not by the other point (Figures 9(a) and 9(b)). Observe that, from the infinite set of circles supported by p_u and, for example, p_i, those intersecting E_{ik} will have p_k as a barrier to their growth. In fact, the largest circle of that set

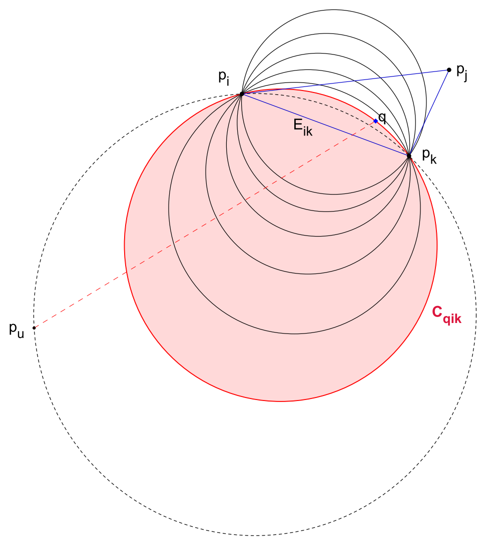


Fig. 8 Some circles supported by both p_i and p_k

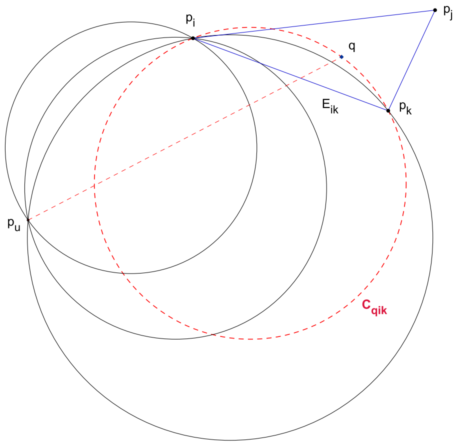
is precisely the one supported by p_i and p_k. As we have just seen in item 1.2.1, a circle supported by p_u, p_i and p_k will never reach q (a contradiction).

1.2.3 The QLEC is supported only by p_u, but not by p_i nor p_k. Then, it must be supported by any other point p. Observe (Figure 10) that, from the infinite set of circles supported by p_u and p, those intersecting E_{ik} will have p_i or p_k as a barrier. In fact, the largest circle of that set is precisely the one supported by p_i (or p_k). As we have just seen in item 1.2.2, a circle supported by p_u and p_i (or p_k) will never reach q (again, a contradiction).

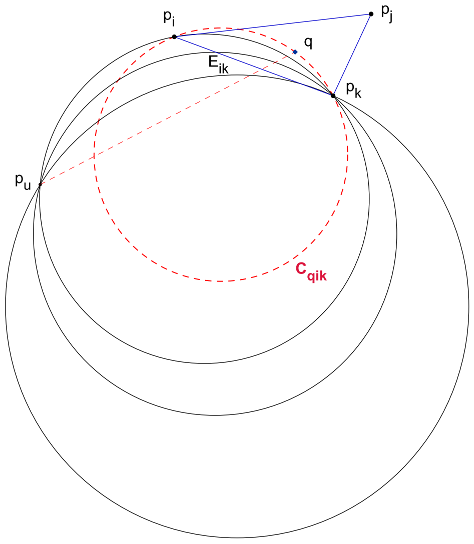
Thus, the initial assumption is false, and p_u must be inside C_{qik} to be able to support the QLEC. The same reasoning can be applied to the other points supporting the QLEC, p_l and p_v.

2. Now, we tackle the case when QLEC is supported by only two points. We know by Lemma 1 that, from all the infinite circles (with different areas) that could be supported by those two points, the QLEC must be the one whose center lies on an edge of $H(S)$. Given that S_q contains, by construction, all the points defining $H(S)$; and given that, as we have already shown, all the points of S supporting the QLEC (two points, in this case: lets assume that p_u and p_v are those points) will also be included in S_q, we can conclude that S_q contains all the points needed to compute the QLEC. □

We have proven that it is safe to use our triangle T_{ijk} in order to filter points. However, there is one more scenario that we must also consider: the situation when it is not possible to find three points of S forming a triangle enclosing q. This occurs when q is outside $H(S)$. Figure 11 shows an example of such a scenario. Nevertheless, even in these circumstances there can be a QLEC having its center within $H(S)$. The next lemma tackles this case.



(a) Some circles supported by both p_u and p_i.



(b) Some circles supported by both p_u and p_k.

Fig. 9 Different circles supported by either p_i or p_k.

Lemma 3 *Let S be a set of points in a fixed axis-parallel rectangle $R \subseteq \Re^2$, and let q be a query point $q \notin S$ such that $q \cap R \neq \emptyset$ and q is outside $H(S)$. Let (p_i, p_j), (p_j, p_k), $\ldots$, (p_n, p_m) be the pairs of points defining all the visible edges of P from q, $V_q(H(S))$. Let S_q be the set of points composed by: the points in C_{qij}, $C_{qjk}, \ldots, C_{qnm}$, and the vertices of $H(S)$. The QLEC obtained from S_q is the same as the one computed from S using q as a query point.*

Proof: We have to prove that with the points inside C_{qij}, $C_{qjk}, \ldots, C_{qnm}$ and those at the vertices of $H(S)$ (which include, $p_i, p_j, p_k, \ldots, p_n, p_m$), we compute the same QLEC (if it exists) as if we compute the QLEC using S.

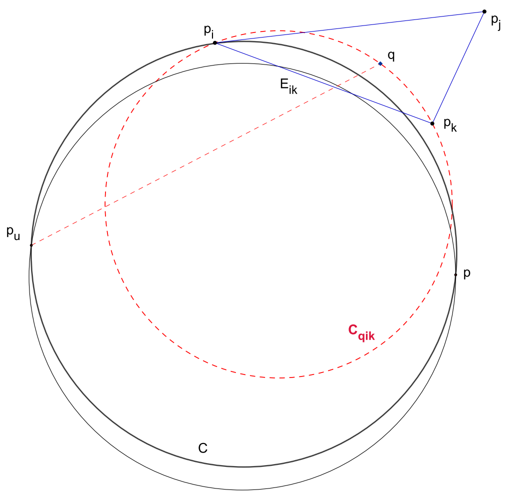


Fig. 10 The circle C is not supported by p_i nor p_k.

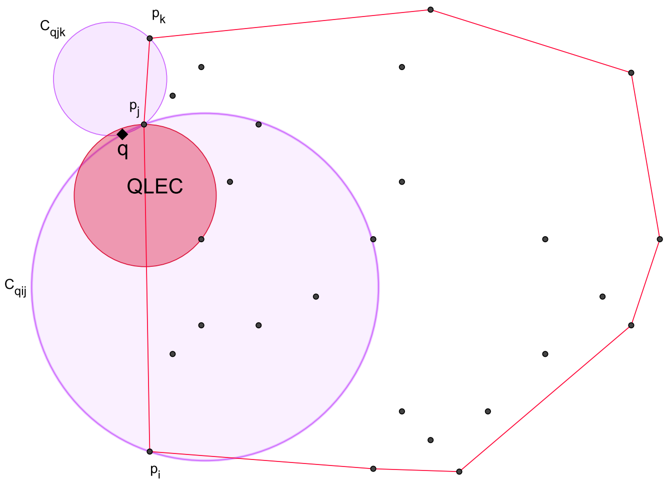


Fig. 11 A query point outside $H(S)$.

By Lemma 1, we know that the QLEC must be supported by two or three points in S, and that the center c of the QLEC must lie inside $H(S)$ or over one of its edges. As q is outside $H(S)$, necessarily the QLEC must intersect $H(S)$.

We are going to prove by contradiction that S_q is enough to compute the QLEC. Let us assume that $p_u \in S$ is one of the points supporting the QLEC, and that S_q does not contain p_u. In particular, that means that p_u must be outside C_{qij} to be excluded from S_q.

Since p_u is not in C_{qij}, we know (Definition 5) that the segment connecting p_u and q must intersect $V_q(H(S))$. Therefore, the QLEC must intersect, or completely contain, at least one of the visible edges in $V_q(H(S))$. Let us assume, without loss of generality, that $E_{ij} \in V_q(H(S))$, defined by the pair (p_i, p_j), is that edge.

Observe that this scenario is exactly the same as those described in Item 1.2. (if three points support the QLEC) or Item 2. (if two points support the QLEC) at the proof of Lemma 2. We have an edge (here, E_{ij}) which separates q and p_u. The QLEC is supported by p_u, and must intersect or completely contain E_{ij} to reach q, with p_i and p_j acting as a barrier for its growth.

So, we are supposing again that p_u is outside C_{qij} (the circle defined by q and the edge vertices, p_i and p_j here), and we can apply the same reasoning used at Lemma 2 to demonstrate that this situation is an absurd. If p_u supports the QLEC, it must be inside C_{qij}. Given that we are including in S_q all the points in C_{qij} and all the vertices of $H(S)$, S_q contains (by construction) all the points needed to find the QLEC. □

Note that in Figure 11, there are two circumferences (C_{qij} and C_{qjk}) passing through two points of the convex hull and the query point. Those circumferences correspond to the visible edges of $H(S)$ from q. The *QLEC*, which contains q and has its center within the convex hull, is supported by p_j (a point of the convex hull) and one other point, both of them in C_{qij}.

5 q−LEC$_{4n}$ algorithm

q−LEC$_{4n}$ can be divided in two phases:

1. The *Filtering phase*, that filters out points of the input data set S in order to obtain a reduced set of points S_q.
   Our method for filtering the input points relies on Lemma 2 (when the query point q is inside $H(S)$) and Lemma 3 (when q is outside $H(S)$).
2. The *computational geometry phase*, that taking as input S_q, runs a classic computational geometry algorithm that obtains the QLEC.

Observe that Lemma 2 and Lemma 3 could be applied directly over the input points without using a spatial index, but this could be really costly. In the case of Lemma 2, the three points defining the triangle enclosing q should be as close to q as possible. This way, the algorithm obtains three circumferences as small as possible, and the set of points passed to the computational geometry algorithm is reduced. However, obtaining close neighbors to q without an index requires a sequential search over the whole set of points. Moreover, it requires to load all the points into memory.

In addition, we need to compute the convex hull of the input set of points. Without the aid of a spatial index, this requires already a computational geometry algorithm like the Graham scan [23] or the divide and conquer algorithm in [38]. These algorithms already have a time complexity $O(n \log n)$ and require to load all the points in main memory, so it would be better to directly apply the computational geometry algorithm that computes the QLEC, which has a $O(n \log n)$ cost.

For these reasons, we take advantage of the spatial indexes defined in Section 3.2 to perform the filtering phase. It is important to notice that the spatial index is maintained by the SDB and directly taken as an input for the algorithm. Thus, the cost of building and maintaining the index is not considered in our work.

Algorithm 1 shows the pseudocode of q−LEC$_{4n}$; next we describe it in detail.

Filtering phase
This phase is composed of the following 3 steps (the first 2 correspond to the case when q is inside $H(S)$ and the third one when it is outside).

Step 1: Using 4NN. Its goal is to obtain, from 4 points close to q, 3 points defining a triangle that contains q. By using a combination of *window query* and *nearest neighbor query* on the spatial index, the algorithm obtains the nearest point with respect to q in each of the four quadrants defined by q (see Definition 4); $S_{nn} = \{nn_{URQ(q)}, nn_{LLQ(q)}, nn_{ULQ(q)}, nn_{LRQ(q)}\}$. Then, the algorithm computes the Delaunay triangulation $(Dn(S_{nn}))$ of S_{nn}.
Figure 12(a) shows the query point q and its 4 nearest neighbors in each quadrant. Their Delaunay triangulation is shown in Figure 12(b). In this example, we can see that there is triangle enclosing q.
However, there can be situations where there is a triangle enclosing q, but it is not obtained by Step 1. For example, q might have no neighbors in one quadrant, as shown in Figure 12(d).
If the triangle is not found in Step 1, we proceed to Step 2.

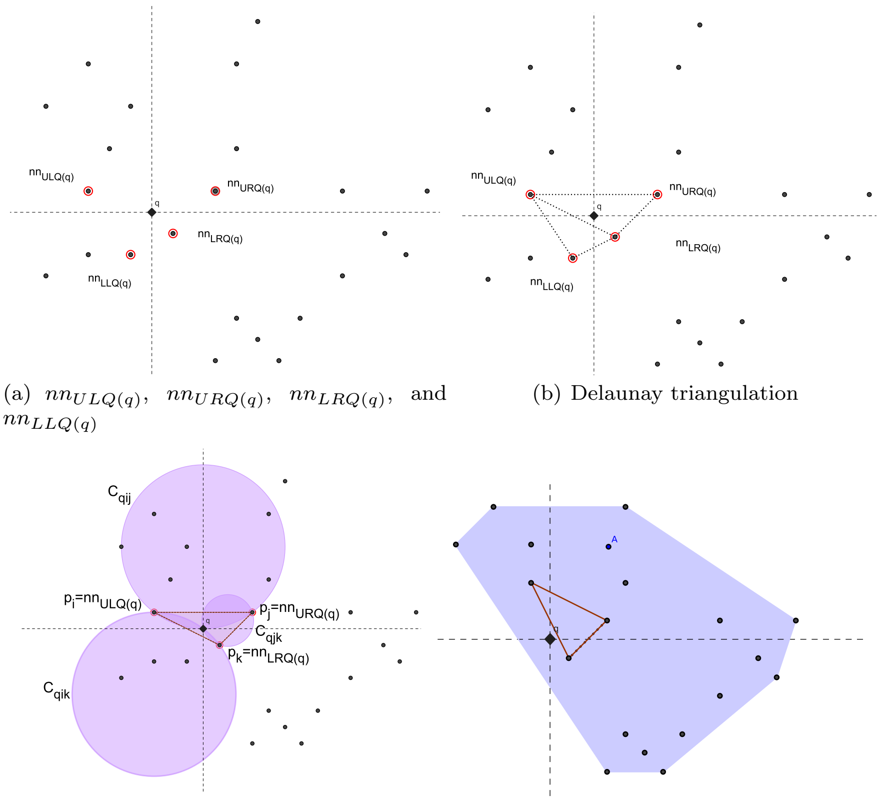
Step 2: Recompute the triangulation including $H(S)$. Add the points defining the vertices $H(S)$ to S_{nn}, and recompute the Delaunay triangulation.
After Steps 1 and 2, if q is within $H(S)$, it is sure that now there is a triangle enclosing q. Let us denote this triangle as T_{ijk}. The algorithm defines the three circles C_{qij}, C_{qik}, and C_{qjk}, using the vertices of the triangle, as shown in Figure 12(c). Only the points inside these circles, plus $H(S)$, need to be fed as input to the computational geometry algorithm.
If the triangle is not found, it means that q is outside the convex hull, and we proceed to Step 3.

Step 3: Use the visible edges of $H(S)$. As shown in Lemma 3, q can be outside the convex hull, and even so, the QLEC can exist. In this case, the algorithm traverses the convex hull looking for pairs of points that define segments *visible* from q (see Definition 5) and adds all the points of S inside the circles supported by q plus each of such pairs of points (see Figure 11).

Computational geometry phase
In this phase, Algorithm 1 adds to S_q all the points that define $H(S)$ (if not already present), and calls the computational geometry algorithm *ComputeLEC* using only these points.

The spatial index (R*-tree or K-D-B-tree in our case) is used to speed up the process, as well as to keep the memory usage low, in several key points: to search for the 4 nearest neighbors of q; to build the convex hull (we use the *depth-first* algorithm of [10]); and to obtain the points inside a circle, filtering out the rest. This last process is shown in Algorithm 2. The logic of this algorithm is independent of the spatial index that is used, as long as it is hierarchical, but the names of the concepts can be different. For this reason, we decided to use the generic name Area to represent a bounding box that encloses a set of points and/or other areas. The concept of Area corresponds to the MBR of an R*-tree, or a Region of a K-D-B-tree.

Algorithm 2 shows how the spatial index is traversed downwards from the root, and processed level by level. At each non-leaf level, those branches whose areas



(a) $nn_{ULQ(q)}$, $nn_{URQ(q)}$, $nn_{LRQ(q)}$, and $nn_{LLQ(q)}$

(b) Delaunay triangulation

(c) C_{qij}, C_{qik}, and C_{qjk}

(d) No triangle from nearest neighbors contains q

Fig. 12 Elements of the q−LEC$_{4n}$ Algorithm.

are completely outside the three circles are no longer considered. At the last level, the algorithm deletes the points inside the leaf nodes that were not filtered by the previous process and that are outside the three circles.

6 Experiments

We have run experiments that execute q−LEC$_{4n}$ coupled with two different spatial indexes, namely, an R*-tree and a K-D-B-tree. The results of these two versions are compared against a *baseline*, which simply loads all points in main memory by reading all the disk blocks that store them, and then solves the problem with a classical computational geometry algorithm.

We used Java as the programming language. For the R*-tree, we used Marios Hadjieleftheriou's Java Implementation.[1] For the K-B-D-tree, we used our own

[1] The author changed his implementation to C++ `http://libspatialindex.github.com/`, however, a slightly modified version of the original Java implementation can be found at `https://github.com/felixr/java-spatialindex`.

Algorithm 1 $q-\mathrm{LEC}_{4n}$

```
1: $q-\mathrm{LEC}_{4n}$(point $q$, multidimensional index $R$)
2: INPUT: $q$ and $R$
3: OUTPUT:$QLEC$ {Largest empty circle only containing $q$}
4: Let $H(S)$=convexHull($R$) {Obtains from $R$ the convex hull of all the points of $S$}
5: for each $x \in (LLQ(q), LRQ(q), URQ(q), ULQ(q))$ do {Step 1}
6:     Let $nn_x$ = computeNearestNeighbor ($q$, $x$, $R$) {Computes the nearest neighbor to $q$ in quadrant $x$}
7: end for
8: Let $S_{nn} = \{nn_{URQ(q)}, nn_{LLQ(q)}, nn_{ULQ(q)}, nn_{LRQ(q)}\}$
9: Let $Dn(S_{nn})$ = computeDelaunay($S_{nn}$)
10: Let $T_{ijk}$ = extractTriangle($Dn(S_{nn})$,$q$) {Extracts the triangle in $Dn(S_{nn})$ that contains $q$}
11: if $T_{ijk} = null$ then {Step 2}
12:     Add the points defining $H(S)$ to $S_{nn}$ {Adds the points at the vertices of $H(S)$}
13:     Let $Dn(S_{nn})$ = computeDelaunay($S_{nn}$)
14:     Let $T_{ijk}$ = extractTriangle($Dn(S_{nn})$,$q$)
15: end if
16: if $T_{ijk}! = null$ then
17:     for all face of $T_{ijk}$ do
18:         Let $p_i$ and $p_j$ the extremes of that face
19:         Let $C_{ij}$ the circle passing through $p_i$, $p_j$, and $q$
20:         Let $S_{ij}$=extractCircle($C_{ij}$,$R$) {Obtains from $R$ the points of $S$ inside $C_{ij}$}
21:         Add $S_{ij}$ to $S_q$
22:     end for
23: else {Step 3}
24:     for all $E_{ij}$ of $H(S)$ {For all edges of the convex hull} do
25:         Let $p_i$ and $p_j$ be the points at the extremes of $E_{ij}$
26:         if $E_{ij}$ is visible from $q$ then
27:             Let $C_{qij}$ the circle passing through $p_i$, $p_j$, and $q$
28:             Let $S_{ij}$=extractCircle($C_{qij}$,$R$)
29:             Add $S_{ij}$ to $S_q$
30:         end if
31:     end for
32: end if
33: Add the points defining $H(S)$ to $S_q$
34: Let $Q_{LEC}$ = ComputeLEC($S_q$, $q$) {The call to the computational geometry algorithm having $S_q$ as input}
35: return $Q_{LEC}$
```

implementation. The computational geometry algorithm (used by both the baseline and $q-\mathrm{LEC}_{4n}$) that computes the LEC is based on Lemma 1 and uses the Java Delaunay Triangulation project[2] by Boaz Ben Moshe that, to compute the Delaunay Triangulation of n points, has a worst case of $O(n^2)$ time complexity, yet in practice, it has an amortized time of $O(n \log n)$, if you shuffle the points.

All tests were run on an isolated Intel®Xeon®-E5520@2.26GHz with 72 GB DDR3@800MHz RAM with a SATA hard disk model Seagate® ST2000DL003-9VT166. It ran Ubuntu® 12.04.5.

We considered the following sets of points:

1. Sets of 100K[3], 250K, 500K, 750K, 1000K, 1250K, and 1500K points with uniform and Gaussian distributions.
2. Three real datasets (see Figure 13): Tiger Streams (ts), Tiger Census Blocks (tcb), and California Roads (ca) datasets, with 194,971 (ts), 556,696 (tcb), and 2,249,727 (ca) points. They were obtained from the *chorochronos* archive.[4]

For all the used datasets, we assumed that all data completely fit in main memory. This is a requirement of the baseline, which simply uses the computational

[2] `https://code.google.com/p/jdt/`
[3] 1K = 1,000 points
[4] `http://chorochronos.datastories.org/?q=node/17`

Algorithm 2 extractCircle

```
1: extractCircle(circle $C$, multidimensional index $R$)
2: INPUT: $C$ and $R$ {a circle and the multidimensional index}
3: Let $E$ and $E_x$ be sets of Areas (MBRs/Regions)
4: Let $P$ be a set of points
5: Let $Area_{root}$ be the Area that covers all the points indexed by $R$
6: Insert the element $Area_{root}$ in $E$
7: Let $l = h - 1$ {$h$ is the height of $R$}
8: while $l \geq 0$ do
9:    if $l > 0$ then
10:       for each $Area_i$ in E do
11:          Substitute $Area_i$ in $E$ by the Areas in the child node corresponding to its entry {the
             node in the next level pointed by the entry containing $Area_i$ is read}
12:       end for
13:       Let $E_x$ the set of Areas of $E$ that are completely outside $C$
14:       Discard from $E$ all the Areas in $E_x$.
15:    else
16:       Let $P = \emptyset$
17:       for each $Area_i$ in E do
18:          Add to $P$ the points in the children of $Area_i$
19:       end for
20:       Remove from $P$ the points outside of $C$
21:       return $P$
22:    end if
23:    Let $l = l - 1$
24: end while
```

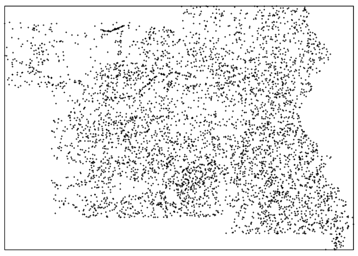
geometry algorithm. Both indexes (R*-tree and K-D-B-tree) were configured to use a disk block size of 1KB. The performance of all algorithms was measured comparing the response time (it represents the overall execution time –elapsed time or wall-clock time– of the algorithms, which is measured in seconds) that each algorithm required to find the solution. The response time includes the time required to read the R*-tree nodes, K-D-B-tree nodes, or points, from disk. For all measures, we computed an average of 1,000 random queries where a solution exists.

6.1 Real data sets

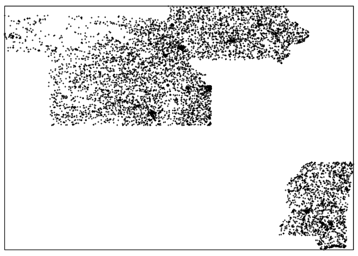
Table 2 shows the running times with the real data sets. The third and fifth columns of the table display the time spent by each of the versions of q−LEC$_{4n}$ (and in parenthesis, the percentage of the global time) during the filtering phase.

	q−LEC$_{4n}$ *alg.*				*Baseline*
	R-tree		KDB-tree		
	Complete	Filtering	Complete	Filtering	Time
ts	0.24	0.15 (63%)	0.26	0.17 (65%)	0.90
tcb	1.21	0.35 (29%)	1.30	0.41 (32%)	3.23
ca	6.33	0.71 (11%)	6.35	0.73 (11%)	28.14

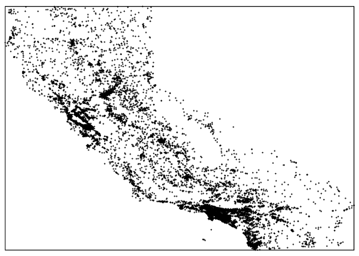
Table 2 Time (seconds) to obtain the answer. In the case of q−LEC$_{4n}$, it is also displayed the time spent during the filtering phase (in parenthesis the percentage of the time to solve the query spent by the filtering phase).



(a) ts



(b) tcb



(c) ca

Fig. 13 The datasets used in the experiments.

Observe that in ca, $q-\mathrm{LEC}_{4n}$ is 4.44 times faster (with the R-tree version) than the baseline, while in tcb the improvement is 2.66 times faster. That difference is due to two factors: the size of the input to the computational geometry phase and the distribution of the points.

The size in the larger datasets has a bigger impact in the computational geometry phase due to its $O(n \log n)$ complexity. Therefore, if we are able to reduce its input size, the gain will be bigger since to reduce the input, our filtering phase basically only pays a logarithmic cost.

Regarding the distribution of the collection, if it gets closer to an uniform distribution, $q-\mathrm{LEC}_{4n}$ runs faster, since it obtains closer neighbors to the query point, and thus C_{qij}, C_{qjk}, and C_{qik} will be smaller (and thus, they allow to filter out much more points).

In ca, we obtain the best improvement (4.44 times), mainly due to the size, since it is by far the biggest data set. With respect to ts and tcb, although tcb is bigger than ts, the distribution of points is more uniform in the case of ts, and thus ts (3.75 times) obtains a better improvement than tcb (2.67 times).

Comparing the results of our algorithm between the different datasets, the percentage of time taken by the filtering phase is bigger in smaller data sets, and decreases as the size of the data set grows. The filtering phase suffers less a growth in the number of input points than the computational geometry phase, given that the $O(n \log n)$ complexity of the computational geometry phase grows faster than the basically logarithmic cost of the filtering phase.

In these data sets, the R-tree version is slightly faster than the K-D-B-tree version, therefore the differences have little impact in the overall running time. In any case, it seems clear that the time spent during the filtering phase is more than compensated when computing the QLEC using fewer points.

From the 1,000 queries, the third column of Table 3 shows the percentage of them solved using each of the three steps of the filtering phase. We can observe that when the data set has larger open areas, this implies that the percentage of queries that were solved using Step 1 decreases. This is easy to see, given that when the query point is located in an open area, there are bigger chances of finding one or more empty quadrants, and also of placing the query point outside the convex hull. Therefore, in ts, 81.3% of the queries were solved using Step 1, whereas in tcb was 55.4% and in ca, 69.2%. The fourth and fifth columns of Table 3 give the average time to obtain the solution, considering the queries solved using each of the three steps of the filtering phase, and for the two versions of q−LEC$_{4n}$. As it can be observed, the improvement in the queries solved using Step 1 is much higher than with the others. For example, with ca and the R-tree version, using Step 1, q−LEC$_{4n}$ is almost 35 times faster than the baseline, whereas in queries solved using Step 3, q−LEC$_{4n}$ is only 18% faster. The reason is easy to see, when q−LEC$_{4n}$ uses Step 1, it is likely that the query point is placed in a dense populated area, and thus that the found neighbors will be close enough to produce small circumferences around the query point. However, in the other cases, the query point is almost in the border of the convex hull (Step 2) or outside the convex hull (Step 3), and thus it is likely that the found neighbors will be quite far away from the query point, and thus the circumferences will be larger. In the ca dataset, we can see another effect that we will observe in the synthetic datasets, the K-D-B-tree version is slightly faster than the R-tree version in zones highly populated, as in the ca dataset when the query is solved using Step 1, that is, when the query is in a dense populated region (as less empty areas are processed).

		Percentage	Time	
			R-tree	KDB-tree
ts	Step 1	81.3%	0.14	0.16
	Step 2	11.6%	0.46	0.49
	Step 3	7.1%	0.99	1.02
tcb	Step 1	55.4%	0.44	0.47
	Step 2	22.8%	1.60	1.70
	Step 3	21.8%	2.78	2.98
ca	Step 1	69.2%	0.81	0.75
	Step 2	15.1%	13.49	13.48
	Step 3	15.7%	23.77	24.06

Table 3 Percentage of queries solved using Step 1, Step 2, and Step 3 of the filtering phase and time (seconds) to obtain the answer considering the queries solved using each of the filtering steps.

We can see in Table 4 the number of points provided as input to both the baseline and the computational geometry phase of q−LEC$_{4n}$. In this table, there is only one value, as the two versions of q−LEC$_{4n}$ obtain the same amount of points. In addition, in parenthesis, it is shown the percentage of points used by q−LEC$_{4n}$ with respect to the total of points (used by the baseline). Again, we can see the effect of the distribution of the points: with the most uniform data set (ts), q−LEC$_{4n}$ uses only 11.24% of the points, but in the tcb data set (the least uniform), the filtering is worse, and thus our algorithm uses 34.52% of the points to compute the QLEC. In addition, we can see again the effect of the step used in the filtering phase. The extreme case is ts, where when the query is solved with Step 1, it is solved using only with 0.88% of the points, whereas, when the query is solved using Step 3, 98.64% of the points are used to solve the query, and thus, as seen in Table 3, q−LEC$_{4n}$ is even slower than the baseline.

	q−LEC$_{4n}$	*Baseline*
ts	21,917 (11.24%)	194,971
Step 1	1,731 (0.88%)	
Step 2	59,084 (30.30%)	
Step 3	192,332 (98.64%)	
tcb	192,215 (34.52%)	556,696
Step 1	45,884 (8.24%)	
Step 2	264,477 (47.50%)	
Step 3	488,506 (87.75%)	
ca	536,862 (23.86%)	2,249,727
Step 1	56,101 (2.49%)	
Step 2	1,329,024 (59.07%)	
Step 3	1,893,997 (84.18%)	

Table 4 Amount of points provided as input to both the baseline and the computational geometry phase of q−LEC$_{4n}$. In parenthesis, the percentage of the points used by q−LEC$_{4n}$ with respect to the total. For q−LEC$_{4n}$, with data disaggregated for the three steps of the filtering phase.

Besides the impact in the running times, reducing the number of points has an important impact on memory consumption. As explained before, the baseline always needs all points of the collection in main memory. Of course, in the case of q−LEC$_{4n}$, accessory space is needed, among other things, to process the spatial index. However, the index never needs to be completely loaded into main memory, and that space is not critical. This effect can be clearly seen in Table 5, where we show the amount of memory (in MBs) used by q−LEC$_{4n}$ and the baseline to compute the final solution. We show the value of the parameter "Maximum resident set size" of the GNU/Linux command "/usr/bin/time". For all values, in parenthesis, it is shown the percentage of memory consumed by q−LEC$_{4n}$ with respect to the baseline. There is a close relationship between the number of points used to solve the query and the amount of main memory used. Therefore, again ts is the data set with smaller percentage of memory consumption with respect to the baseline, and tcb the worst one.

The data disaggregated by the filtering step shows that, in the case of ts and Step 3, q−LEC$_{4n}$ uses 98.64% of the points used by the baseline, and then, the R-tree version uses the 99.31% of the main memory consumption of the baseline. However, with Step 1, while q−LEC$_{4n}$ uses only the 0.88% of the points, the

R-tree version uses 16.54% of the main memory wasted by the baseline. This non-proportional decrease is probably due to the chosen programming language. Java does not allow the programmer to release the memory wasted by unused variables. Instead, its running environment triggers a "garbage collector" (GC) that is responsible for releasing the memory reserved for variables that are no longer used. However, an unused variable or object that maintains a valid reference will not be released. Additionally, the GC is run when the running environment decides to do that. The frequency of execution of the GC depends mainly on the available free memory, generating a lack of control. In our case, after finishing the filtering phase, all used data structures could be released, and the memory could be reused by the computational geometry phase. However, the values clearly show that those structures are not released by the GC, as in the aforementioned case of ts with Step 1.

Observing Table 5, the next question is why the filtering phase of the K-D-B-tree version consumes more memory than the R-tree version. The reason is that during the searches, usually the K-D-B-tree version has to keep more regions in a data structure (a stack or a sorted heap) to be inspected. For example, when using the algorithm of Böhm and Kriegel to compute the convex hull with the aid of the spatial index, the first step requires the points with the maximum values in the x and y coordinates and the points with the minimum x and y coordinates. Observe that, for example, to obtain the point with the maximum value in the x coordinate, when using the R-tree version, the algorithm starts the search from the root of the tree by choosing the MBR in that node that has the right side more to the right, since in the R-tree it is sure that there is a point in that side. Then the algorithm checks the children of that entry, and again, from those MBRs, that with the right side more to the right is chosen, and so on. Therefore, during the search, only one MBR is kept. However, in the case of the K-D-B-tree, the sides of the regions are not so helpful, since many regions can cover the space up to the limits of the space, and there is not guarantee that a point will be in the sides of the region. This means that the search must keep several regions in a sorted heap to be processed in subsequent steps.

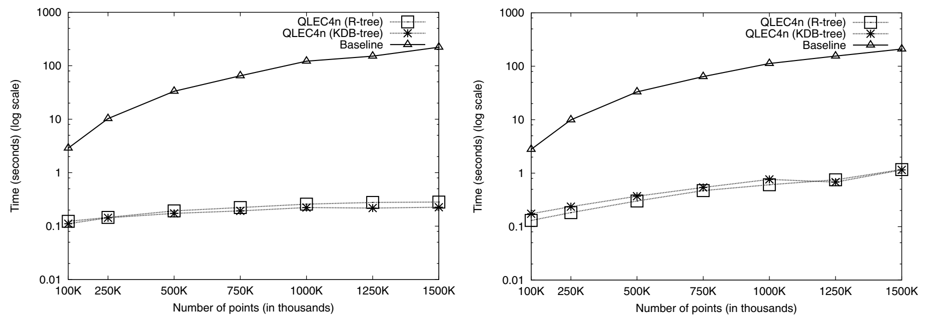
6.2 Synthetic data sets

Figure 14 shows the behavior of the algorithms as the size of the collection grows, considering the synthetic collections. The y axis shows the time using a logarithmic scale to fit the times of the baseline in the same figure. The x axis shows the number of points of the collection.

With a uniform distribution, the R-tree version of $q-\mathrm{LEC}_{4n}$ is between 23 and 786 times faster than the baseline. The K-D-B-tree version is between 25 and 981 times faster. These values can be misleading (the K-D-B-tree seems much better), but differences between the two versions are small. With the input dataset of 1,500,000 points, the times of the two versions only differ in 0.06 seconds. Considering the Gaussian distribution, as expected, the improvements are smaller, from 16 up to 227 times faster. The reasons are those already pointed out for the real datasets. The uniform distribution allows to obtain closer neighbors to the query point. Moreover, from the 7,000 random queries run in this experiment (1,000 for each data set), with the uniform distribution only one query was solved

	$q-\mathrm{LEC}_{4n}$		*Baseline*
	R-tree	KDB-tree	MBs
ts	222 (25.24%)	358 (40.78%)	878
Step 1	145 (16.54%)	259 (29.47%)	
Step 2	359 (40.90%)	545 (62.05%)	
Step 3	872 (99.31%)	1,189 (135.50%)	
tcb	778 (45.56%)	1,340 (78,54%)	1,707
Step 1	310 (18.17%)	607 (35.55%)	
Step 2	1,076 (63.06%)	1,805 (105.76%)	
Step 3	1,459 (85.45%)	2,413 (141.37%)	
ca	1,903 (26.19%)	3,373 (46.41%)	7,267
Step 1	310 (4.27%)	844 (11.61%)	
Step 2	3,518 (48.40%)	5,105 (70.24%)	
Step 3	3,447 (47.44%)	6,095 (83.87%)	

Table 5 Memory consumption (MBs), in parenthesis the percentage of the memory used by $q-\mathrm{LEC}_{4n}$ with respect to the baseline. For $q-\mathrm{LEC}_{4n}$, with data disaggregated for the three steps of the filtering phase.



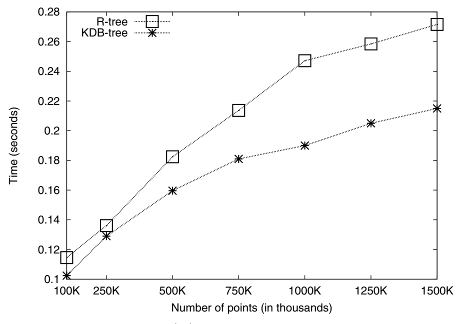
(a) Uniform (b) Gaussian

Fig. 14 Time (seconds) to compute the solution as the size of the collection grows, with synthetic collections. The y axis is in logarithmic scale.

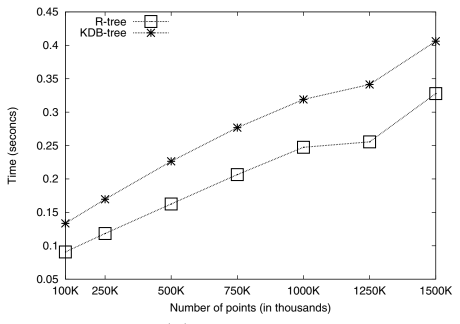
using Step 2 and none using Step 3. However, with Gaussian distribution 71.5% were solved using Step 1, 5.19% using Step 2, and 23.31 % using Step 3 and, as shown in the experiments with real data sets, the improvements in queries solved with steps 2 and 3 are worse.

Figure 15 shows the time spent by the two versions of $q-\mathrm{LEC}_{4n}$ in the filtering phase. The uniform distribution is the best scenario for $q-\mathrm{LEC}_{4n}$, and among the two versions, the K-D-B-tree one is better here. Figure 15(a) shows this superiority. Observe that the gap between the two versions, although small in all cases, gets bigger as the collection grows. The reason is that all data sets have their points in the rectangle $[0, 0] \times [1, 1]$. As the size of the collection grows, the density gets higher, and this favors the K-D-B-tree version. In the Gaussian distribution (Figure 15(b)), we can see the opposite behavior just like in real data sets, where the R-tree version is better than the K-D-B-tree.

Figure 16 shows the number of points provided as input to the computational geometry phase of $q-\mathrm{LEC}_{4n}$, and to the baseline. The y axis has a logarithmic



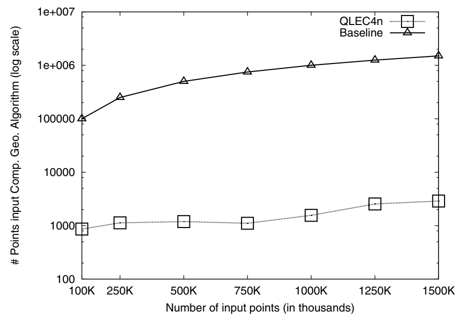
(a) Uniform



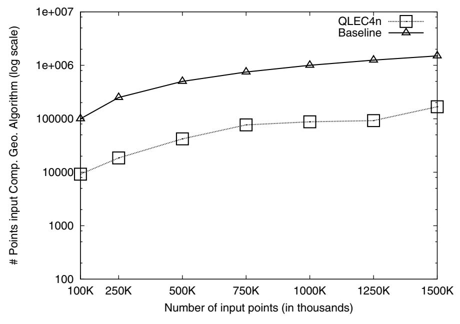
(b) Gaussian

Fig. 15 Time (seconds) spent during filtering phase as the size of the collection grows, with synthetic collections.

scale in order to be able to fit the values of the baseline. Again, in this figure, we can observe that q−LEC$_{4n}$ is more successful with uniform distributions: to obtain the final answer, in the case of the uniform distribution, our algorithm uses between 0.15% and 0.87% of the input points, whereas in the case of the Gaussian distribution, our algorithm uses between 7.38% and 11.19% of the input points.



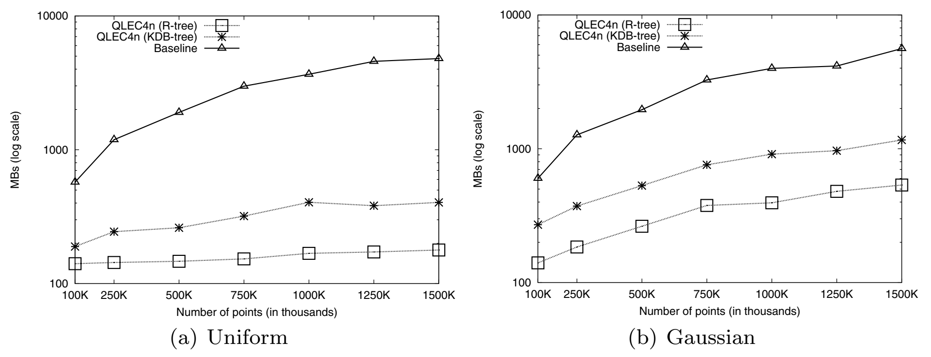
(a) Uniform



(b) Gaussian

Fig. 16 Points provided as input to the computational geometry phase of q−LEC$_{4n}$, and to the baseline. The y axis is in logarithmic scale.

Finally, Figure 17 shows the memory consumption. Again, the y axis has a logarithmic scale to fit the values of the baseline. This figure follows the same trend of previous experiments. The results with the uniform distribution are better, with the R-tree version, q−LEC$_{4n}$ uses between 3,71% and 24.58% of the memory used by the baseline, whereas with the Gaussian distribution, the values range between 9.56% and 23.36%. As seen with the real data sets, the R-tree version consumes much less memory, in most cases around 50% of the memory consumption used by the K-D-B-tree version.



(a) Uniform (b) Gaussian

Fig. 17 Memory consumption as the size of the collection grows, with synthetic collections. The y axis is in logarithmic scale.

7 Conclusions

In this work, we have presented an algorithm to efficiently solve the problem of finding the largest empty circle containing only a query point q in spatial databases. Previous solutions to solve this problem were static in-memory solutions, whereas our solution relies on typical spatial indexes which can work without the need to store the complete structure in main memory and have the ability to dynamically adapt their structures to changes in the stored data.

With this algorithm, we open a way for including a new query for spatial database management systems, since the new algorithm, in addition to improved running times, consume much less memory than the previous computational geometry algorithms.

Our experiments show that our algorithm requires much less response time than the original computational geometry algorithm approach: with real datasets, in the range of 2.5-4.4 times faster; and with synthetic datasets, 23-981 times faster (uniform distributions) and 16-227 times faster (Gaussian distributions). Since, in the real datasets, our algorithm uses 11.24%-34.52% of the points used by the computational geometry algorithm, the $q-\mathrm{LEC}_{4n}$ consumes only between 25.24% and 78.54% of the main memory consumed by the computational geometry algorithm. In the case of synthetic data sets, $q-\mathrm{LEC}_{4n}$ uses between 0.15% and 0.87% (uniform distributions) and between 7.38% and 11.19% (Gaussian distributions) of the points used by the computational geometry algorithm, and thus the memory consumption is between 3.71% and 33.02% (uniform distributions) and between 9.56% and 45.03% (Gaussian distributions) of the main memory consumed by the computational geometry algorithm.

We also showed that our algorithms scale much better than the computational geometry algorithm with synthetic datasets.

As future work, we want to extend our proposal to objects with more dimensions.

References

1. Aggarwal, A., Suri, S.: Fast algorithms for computing the largest empty rectangle. In: Proceedings of Third Annual Symposium on Computational Geometry SCG 1987, pp. 278–290 (1987)
2. Augustine, J., Das, S., Maheshwari, A., Nandy, S.C., Roy, S., Sarvattomananda, S.: Querying for the largest empty geometric object in a desired location. CoRR **abs/1004.0558v2** (2010)
3. Augustine, J., Das, S., Maheshwari, A., Nandy, S.C., Roy, S., Sarvattomananda, S.: Localized geometric query problems. Computational Geometry **46**(3), 340 – 357 (2013)
4. Augustine, J., Putnam, B., Roy, S.: Largest empty circle centered on a query line. Journal Discrete Algorithms **8**(2), 143–153 (2010)
5. Augustine, J., Putnam, B., Roy, S.: Largest empty circle centered on a query line. Journal of Discrete Algorithms **8**(2), 143–153 (2010)
6. Azri, S., Ujang, U., Anton, F., Mioc, D., Rahman, A.A.: Review of spatial indexing techniques for large urban data management. In: Proceedings of International Symposium & Exhibition on Geoinformation ISG 2013 (2013)
7. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. Acta Informatica **1**(3), 173–189 (1972)
8. Beckmann, N., Kriegel, H., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of ACM SIGMOD Conference on Management of Data, pp. 322–331 (1990)
9. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM **18**(9), 509–517 (1975)
10. Böhm, C., Kriegel, H.P.: Determining the convex hull in large multidimensional databases. In: Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery DaWaK 2001, pp. 294–306 (2001)
11. Bose, P., Wang, Q.: Facility location constrained to a polygonal domain. In: Proceedings of the Latin American Symposium on Theoretical Informatics LATIN 2002, pp. 153–164. Springer (2002)
12. Chaudhuri, J., N., S.C., Das, S.: Largest empty rectangle among a point set. Journal Algorithms **46**, 54–78 (2003)
13. Cheng, L., Wu, C., Zhang, Y., Wang, Y.: An energy-balance repair scheme in wireless sensor networks. Journal of Information and Computational Science **8**(6), 969–976 (2011)
14. Cheng, M.X., Li, D.: Advances in Wireless Ad Hoc and Sensor Networks. Springer, New York, NY, USA (2008)
15. Chew, L.P., Drysdale, R.L.S.: Finding largest empty circles with location constraints. Tech. Rep. PCS-TR86-130, Dartmouth College, Computer Science, Hanover, NH (1986)
16. Corral, A.: Algoritmos para el procesamiento de consultas espaciales utilizando r-trees. la consulta de los pares más cercanos y su aplicación en bases de datos espaciales. Ph.D. thesis, Universidad de Almería, Escuela Politécnica Superior, España (2002)
17. Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M.: Algorithms for processing k-closest-pair queries in spatial databases. Data & Knowledge Engineering **49**(1), 67–104 (2004)
18. Dumitrescu, A., Jiang, M.: Computational geometry column 60. SIGACT News **45**(4), 76–82 (2014)
19. Edmonds, J., Gryz, J., Liang, D., Miller, R.J.: Mining for empty spaces in large data sets. Theoretical Computer Science **296**, 435–452 (2003)
20. Filipe, L., Vieira, M., Augusto, M., Vieira, M., Ruiz, L.B., Alfredo, A., Loureiro, F., Silva, D.C., Otviofernandes, A., Carlos, A.A., mg Brazil, P.B.H.: Efficient incremental sensor network deployment algorithm. In: Proceedings of Brazilian Symposium on Computer Networks SBRC 2004 (2004)
21. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta Informatica **4**(1), 1–9 (1974)
22. Gargantini, I.: An effective way to represent quadtrees. Communications of the ACM **25**, 905–910 (1982)
23. Graham, R.: An efficient algorithm for determining the convex hull of a finite planar set. Information Processing Letters **1**, 132–133 (1972)
24. Gutiérrez, G., Paramá, J.R.: Finding the largest empty rectangle containing only a query point in large multidimensional databases. In: Proceedings of Conference on Scientific and Statistical Database Management SSDBM 2012, pp. 316–333 (2012)

25. Gutiérrez, G., Paramá, J.R., Brisaboa, N., Corral, A.: The largest empty rectangle containing only a query object in spatial databases. GeoInformatica **18**(2), 193–228 (2014)
26. Gutierrez, G., Sáez, P.: The k closest pairs in spatial databases - when only one set is indexed. GeoInformatica **17**(4), 543–565 (2013)
27. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD Conference on Management of Data, pp. 47–57 (1984)
28. Hjaltason, G.R., Samet, H.: Incremental distance join algorithms for spatial databases. In: Proceedings of ACM SIGMOD Conference on Management of Data, pp. 237–248 (1998)
29. Kaminker, T., Sharir, M.: Finding the largest disk containing a query point in logarithmic time with linear storage. In: Proceedings of the Thirtieth Annual Symposium on Computational Geometry SCG 2014, pp. 206:206–206:213 (2014)
30. Kaplan, H., Mozes, S., Nussbaum, Y., Sharir, M.: Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In: Proceedings of Symposium on Discrete Algorithms SODA 2012, pp. 338–355 (2012)
31. Kaplan, H., Sharir, M.: Finding the maximal empty disk containing a query point. In: Proceedings of the Twenty-eighth Annual Symposium on Computational Geometry SCG 2012, pp. 287–292 (2012)
32. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer-Verlag, London, UK (2006)
33. Mellou, K.: Efficient algorithms for calculating the maximum empty cube in areas with obstacles. Ph.D. thesis, National Technical University of Athens, Greece (2014)
34. Naamad, A., Lee, D., Hsu, W.L.: On the maximum empty rectangle problem. Discrete Applied Mathematics **8**(3), 267 – 277 (1984)
35. O'Rourke, J.: Art Gallery Theorems and Algorithms. Oxford University Press, Inc., New York, NY, USA (1987)
36. O'Rourke, J.: Computational Geometry in C. Cambridge University Press, Cambridge, UK (1998)
37. Preparata, F., Shamos, M.: Computational geometry: an introduction. Springer-Verlag, New York, NY, USA (1985)
38. Preparata, F.P., Hong, S.J.: Convex hulls of finite sets of points in two and three dimensions. Communications of the ACM **20**(1), 87–93 (1977)
39. Robinson, J.T.: The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 10–18 (1981)
40. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. SIGMOD Record **24**(2), 71–79 (1995)
41. Samet, H.: Foundations of Multidimensional and Metric Data Structures. MorganKaufmann, San Francisco, CA, USA (2006)
42. Shamos, M.I.: Computational geometry. Ph.D. thesis, Dept. Computer Sciences, Yale University (1978)
43. Shamos, M.I., Hoey, D.: Closest-point problems. In: Proceedings of 16th Annual Symposium on Foundations of Computer Science FOCS 1975, pp. 151–162. IEEE (1975)
44. Stratil, H.: An efficient implementation of the greedy forwarding strategy. In: Proceedings of Informatik 2004, Informatik verbindet, Band 2, Beitrge der 34. Jahrestagung der Gesellschaft fr Informatik e.V., pp. 365–369 (2004)
45. Toussaint, G.: Computing largest empty circles with location constraints. International Journal of Computer and Information Sciences **12**(5), 347–358 (1983)
46. Valentine, F.A., Buchman, E.: External visibility. Pacific Journal of Mathematics **64**(2), 333–340 (1976)
47. Wu, C.H., Lee, K.C., Chung, Y.C.: A delaunay triangulation based method for wireless sensor network deployment. Computer Communications **30**(14-15), 2744–2752 (2007)