Closest-point queries for complex objects

Eugene Greene* and Asish Mukhopadhyay†
School of Computer Science
University of Windsor
Windsor, Ontario
Canada

Abstract

In this paper we report on the implementation of a heuristic for computing the closest of a set of n given points in the plane to a complex query object, to wit a triangle, a circle, a rectangle etc. Our results indicate that the heuristic is effective for query objects with small perimeter relative to the given point set, and large values of n.

1 Introduction

There is a vast literature on the problem of preprocessing a set of n sites $S = \{p_1, p_2, p_3, \ldots, p_n\}$ to find the site that is closest to a query point p (see Clarkson [4], for example). The problem that has been less studied is that of preprocessing the set of sites to answer a proximity query for an object that is more complex than a simple point, to wit, a line, a half-line, a line-segment, a rectangle, a convex polygon with a fixed number of sides, a circle, or an ellipse. Assuming that the distance from a point to a query curve can be determined in constant time, then a query can be answered in $O(n)$ time, using $O(n)$ space to store the points in a list. The problem becomes difficult if we want sublinear query time. In this paper we address this problem.

Mitra and Chaudhuri [11] discuss this problem, for lines, partly because it can appear in pattern classification and data clustering. Other applications include those in geographic information systems. It is easy to see applications of this problem when the query object is considered as a path followed by some object.

One motivation for this paper, from a theoretical point of view, is to show the connection of this problem with the very well-studied range-query problem in Computational Geometry. However, the data structures that one would borrow from range-query problems are extremely complicated, and this brings us to the other goal of this paper - is there a simple way of doing this that can be easily implemented in practice? Indeed there is, and we show a simple solution that can be uniformly applied to a variety of different query objects.

The paper is organized as follows. In the next section we discuss some prior work. In the following section we briefly introduce the partition tree data structure, and show a general connection between range queries and closest-point queries in the section after that. In the fifth section, we outline our heuristic technique, and we conclude in the sixth section.

2 Prior Work

One of the first papers to address this problem when the query object is a line was by Cole and Yap [5]. They reported a solution with preprocessing time and space in $O(n^2)$ and query time in $O(\log n)$. Lee and Ching [8] obtained the same result using geometric duality. Mitra [10] reported an algorithm with preprocessing time

*greene6@uwindsor.ca; Supported by an NSERC USRA
†asishm@cs.uwindsor.ca; Partially supported by an NSERC operating grant

and space in $O(n \log n)$ and query time in $O(n^{0.695})$. In a subsequent paper, Mitra and Chaudhuri [11] improved the space complexity to $O(n)$. In a paper by Mukhopadhyay [13], the simplicial partition technique of Matousek [9] was used to improve the query time to $O(n^{1/2+\epsilon})$ for arbitrary $\epsilon > 0$, with preprocessing time and space in $O(n^{1+\epsilon})$ and $O(n \log n)$ respectively. Nandy et al [14] reported an algorithm for the k-nearest neighbours of a query line with preprocessing time and space in $O(n^2)$ and $O(n^2/\log n)$ respectively, and query time in $O(k + \log n)$.

When the query object is a line segment, Goswami et al [7] reported an algorithm for computing the k-nearest points with both preprocessing time and space in $O(n^2)$ and query time in $O(k + \log^2 n)$. The solution is reduced to two triangle range queries and to two point-locations in a k-th order Voronoi diagram and involves some complicated data structures. It is not a surprise therefore that no implementation was reported. It is straightforward to extend the solution of that paper to the case when the query object is a half-line.

When the query object is a circle, Mitra et al [12] reported two different schemes: one of these has preprocessing time and space in $O(n^3)$ and query time in $O(\log^2 n)$; the other has preprocessing time and space in $O(n^{1+\epsilon})$ and $O(n \log n)$ respectively and query time in $O(n^{2/3+\epsilon})$. No implementation was reported for the algorithm in this paper either. The latter scheme uses a lifting transformation that, it turns out, is not required if we make use of the partition trees of the next section. These trees reduce the complexity of a query to $O(\sqrt{n}\ polylog(n))$, but increase the preprocessing time to a larger polynomial.

A related problem is that of continuous nearest neighbor: given some query curve, partition the curve into intervals so that each point in a given interval has the same closest site, and report each such interval and closest-site pair. Tao et al [15] used R-trees for the continuous nearest neighbor problem for line segments, while De Almeida [6] used R*-trees for another implementation.

3 Partition Trees

Welzl [17] introduced a partition tree data structure to solve range query problems. His solution relied on ideas introduced by Vapnik and Chervonenkis [16]. This solution has an intimate connection with our problem, and to bring this out, we briefly discuss the range-query problem in a slightly formal setting. A range space is a pair (X, R), where X is a non-empty set and R is a set of subsets of X, called ranges. For example, let X be the two-dimensional Euclidean space and R the set of open half-planes. The range query problem is to determine the set of points of a finite subset S of X that lie in a given query range $r \in R$.

For the example range space above, if S is a set of three points in the plane then every subset of S is the result of intersecting S with some half-plane. We say that S is shattered by (X, R). However, if S is any set of four points in the plane then we can always find a subset B for which there is no half-plane H such that $H \cap S = B$. A range space (X, R) has finite VC dimension d, if d is the cardinality of the largest set S that can be shattered by (X, R). If no such d exists, then (X, R) is said to have infinite VC dimension.

In an important paper, Welzl [17] showed that if a range space has a finite VC dimension then the points of S can be organized into a partition tree (see Fig. 1) so that each range query visits few (sublinear in $n = |S|$) nodes of the partition tree. For a given range r, at each node of the partition tree, we have to resolve if the set of points (of S) at this node is *contained* in the range, is *disjoint* from it, or *crosses* it. In the first case we include the points for counting or reporting; in the second case we exclude the points completely. But for both cases the search stops at this node. In the third case, the query continues with the children of the node.

To come to our problem, consider cases in which the closest-point query object happens to be the boundary of a range (for example, the boundary of a half-plane is a line, that of a disk is a circle) from a range space of finite VC dimension. Once we have determined that a set of points at a node is on either side of this boundary, then we can structure these points so that the closest-point query can be answered efficiently. Thus Mitra and Chaudhuri [11] and Mukhopadhyay [13] organized the points into a convex hull (in fact, this has to be done for range queries also). Mitra et al's [12] algorithm, examining query circles, maintains the

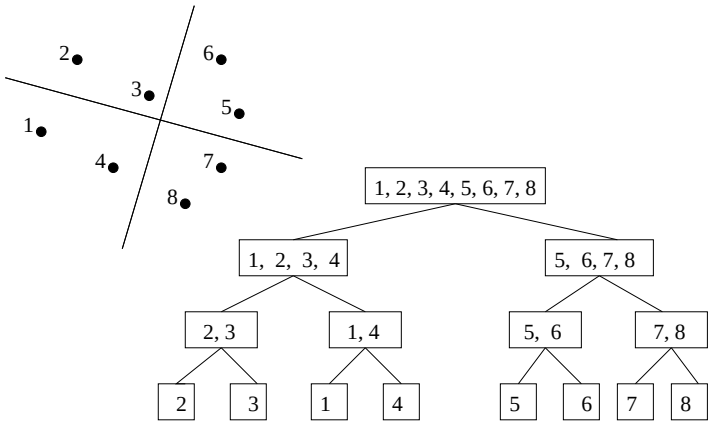

Figure 1: *An example partition tree*

nearest and furthest point Voronoi diagrams at each node.

This correspondence between range and closest point queries is helpful whenever it is beneficial to group points on either side of the query curve. The only problem then becomes finding the closest points to the query on either side of it, which might be a nontrivial task. This happens when, for example, the query object is an ellipse, a rectangle, or a convex polygon with a fixed number of sides, even though these are boundaries of ranges in range spaces of finite VC dimension. Let us elaborate on this by taking the example of the ellipse.

Here the first issue is to define how we measure the distance from a point p to the boundary of the ellipse. We could define it in terms of a point x on the ellipse such that $\overline{px}$ is normal to the ellipse at x. This, however, doesn't use any of the fixed points of the ellipse like its center or foci. Instead, we can define the distance as the difference $|2a - dist(p, f_1) - dist(p, f_2)|$, where $2a$ is the length of the major axis of the query ellipse, while $dist(p, f_1)$ and $dist(p, f_2)$ are the distances from p to the foci f_1 and f_2 of the query ellipse. This essentially reduces our problem to answering the following question for a query segment: find a point of S outside (respectively inside) the ellipse such that the sum of its distances from the end-points of the query segment is a minimum (maximum). No matter how the distance is defined, identifying a set of points that lies entirely inside or entirely outside the ellipse, does not help us in choosing a point from this set that is closest to the boundary in an obvious manner.

4 General tree-based queries

Welzl's partition trees can not be used when the query object is not the boundary of a range from a range space of finite VC dimension. Consider for example a line segment or a half-line. These have to be dealt with in a manner as shown, for example, by Goswami et al [7]. Depending on the type of query objects, it could be beneficial to use other tree structures, such as Matousek's [9] simplicial partition trees, that put some bound on the number of nodes visited.

Consider a range query algorithm, based on any partition tree, of the form in Figure 2. Algorithm *RangeQuery* will determine the intersection of *range* with the points associated with some *node* of a partition tree. Then, after possibly adding some information to each node of the tree, this algorithm can be modified to a closest-point query algorithm. See Figure 3. The queries for Algorithm *ClosestPointQuery* are the boundaries of the ranges in Algorithm *RangeQuery*. The extra information at each node would be required for steps 2(1)1 and 2(2)1. Also, the algorithm might need two different techniques for steps 2(1)1 and 2(2)1.

Algorithm *RangeQuery*$(node, range)$

1. If *node* is a leaf, then use brute force to return *answer*
2. For each *child* of *node*
    1. If *child*'s points are outside *range*, then disregard *child*
    2. Else if *child*'s points are inside *range*, then add *child*'s points to *answer*
    3. Else add *RangeQuery*$(child, range)$ to *answer*
3. Return *answer*

Figure 2: *A general tree-based range query algorithm*

Algorithm *ClosestPointQuery*$(node, curve)$

- *curve* divides the plane $\mathbb{R}^2$ into *range* and $\mathbb{R}^2 - range - curve$

1. If *node* is a leaf, then use brute force to return *answer*
2. For each *child* of *node*
    1. If *child*'s points are outside *range*, then
        1. Find the point p of *child* that is closest to *curve*
        2. If p is closer to *curve* than *answer* is, then update *answer*
    2. Else if *child*'s points are inside *range*, then
        1. Find the point p of *child* that is closest to *curve*
        2. If p is closer to *curve* than *answer* is, then update *answer*
    3. Else
        1. $p = ClosestPointQuery(child, curve)$
        2. If p is closer to *curve* than *answer* is, then update *answer*
3. Return *answer*

Figure 3: *A general tree-based closest point query algorithm*

5 A heuristic approach

The above data structures proposed for range queries are interesting in principle but are also complicated. The construction of the partition trees relies on a range of techniques, and each node of the tree could require three or more other data structures on the points the node contains. To the best of our knowledge, we are not aware of any implementations of these data structures. CGAL's [2] (Computational Geometry Algorithms Library) range query data structure, for example, is a delaunay triangulation that uses a depth-first search to answer queries.

Here we propose a practical heuristic for answering closest point queries, for curves in the plane, that is extremely amenable to easy implementation and that works uniformly for many kinds of query objects. This method also works for the continuous nearest neighbor problem. We construct a Voronoi diagram of the input point set and restrict this diagram to a large (implicit) bounding box that encloses all the points and all potential query objects. In addition, we require a point-location structure on this diagram. Given a query curve, we pick any point on the curve and locate the Voronoi cell containing this point. Now we traverse the diagram along the query curve. We can easily prove that the a query object will cross the Voronoi cell corresponding to the site that is closest to the query object.

One of the main advantages of this approach is that the Voronoi diagram is a very common data structure, and implementations already exist in many libraries and/or languages. A system (a GIS for example) may even already be using a Voronoi diagram to store sites, in which case this technique requires little or no additional initialization time and requires little additional space.

This technique allows for query curves that satisfy the following conditions:

1. It is possible to pick some initial point on the curve
2. It is possible to determine the distance from a point to the curve
3. Given some starting point p on the curve, and a (possibly unbounded) polygon containing p, it is possible to determine the first edge (or vertex) of the polygon that the curve intersects, or to determine that the curve does not exit the polygon
4. It is possible to determine the direction of the tangent to the curve at a point
5. It might be necessary to be able to determine the point of intersection between the curve and a polygon edge (i.e. lines, rays, and segments)

5.1 Implementation Details

This walk-based idea was implemented using CGAL [2]. CGAL's Voronoi diagram data structure provides both point location and the Voronoi cells through which query objects are followed. The main computational problem is this: a part of the query curve is in some cell in the diagram (after possibly having entered the cell through some edge); then through which cell edge does this part of the query curve exit the cell? Four approaches to this problem were tested.

Counter-clockwise edge traversal Starting at some (possibly arbitrary) edge of a Voronoi cell, we travel counter-clockwise around the interior of the cell, until we hit an edge that intersects the query object. This method works when the query curves can be broken into sections, none of which makes a right turn. The query object might need to be broken down further, for example in the case of a spiral.

Binary search This technique only works for linear queries. We assume that the query object has entered the current Voronoi cell through some initial edge. (The implementation of this method actually uses the previous method to exit the very first cell we encounter.) This is not absolutely necessary, it is just simpler. We perform a binary search on the edges of the cell to find the one that intersects the query object. Assuming the query curves can be broken into lines, half-lines, and segments, this method would seem beneficial. If a Voronoi diagram implementation does not support random access on edges of a cell (which is the case in CGAL's structure), then additional preprocessing and space is required to allow a binary search.

Radial triangulation traversal Every Voronoi cell is triangulated, using the cell's site as a central vertex, to create one triangle for each edge. The motivation for introducing triangulations is that, given triangular cells, it is easier to determine the edge through which a query curve exits a cell. Note that this triangulation need not be explicit. This triangulation depends only on individual Voronoi cells, and so it can be used implicitly.

Random triangulation traversal Every Voronoi cell is randomly triangulated without adding any vertices. The implementation converted Atkinson and Sack's binary trees [3] into triangulations for this purpose. This triangulation has fewer triangles than the previous one. On the other hand, it might be that a query curve will end up intersecting more edges in this triangulation. Extra preprocessing time is required to add triangulation edges to a Voronoi diagram.

Whenever we encounter an edge intersecting the query curve, we jump over the edge into the next cell, and repeat the process of exiting the cell. For closed curve queries, we need to keep track of the starting point

of the walk so that we stop when reaching it again. The implementation stores the first edge that the query intersects, and also the corresponding intersection point.

It could be that the query curve passes through a vertex of the diagram. In this case, we use the tangent of the curve (at that vertex) to determine the cell that the curve enters. If no more than 3 sites are ever on the same circle, then each Voronoi vertex will have exactly 3 adjacent edges, and it is straightforward to determine how the curve exits the vertex. If it is possible that Voronoi vertices can have many adjacent edges, then we can perform a binary search on the adjacent edges to find the exit.

5.2 Experimental Details

The implementation was tested against a brute-force algorithm (looking at each site and checking for the minimum distance). The tests involved circle, triangle, and rectangle queries. In addition to query times, the initialization times were recorded (see Fig. 5). For comparison, the initialization times of the Voronoi diagram are listed with those of CGAL's range query structure. The range query structure is based on a delaunay triangulation, and so the Voronoi diagram structure is actually constructed from the range query structure.

There are three potential variables for each query: the number of sites, the span of the sites, and the perimeter of the query object with respect to the site span. The tests vary two parameters: the number of sites per set, and the perimeter of query objects with respect to the span of site sets; the span of the sites was kept at a constant 1000 units. If a query object has a relative perimeter of 1/2, then it's perimeter is approximately half the vertical (and horizontal) distance spanned by the site set.

The test system was as follows: 2 GHz AMD Athlon 64 X2 3800+ processor; 896 MB RAM; running under Windows XP Professional x64; compiled by Visual C++ 2005 Express. Site sets and query objects were randomly generated using a uniform distribution from the Boost library [1]. See Figure 5 for initialization times. This test generated a new point set for a number of different set sizes. Figures 6, 10, and 14 show how varying the number of sites affects the query time. Each of these tests kept a fixed query object, and used the site sets generated in the previous test. Figures 7 to 9, 11 to 13, and 15 to 17 show how varying the query perimeter affects the query time. Each of these tests used ten different site sets, and generated ten different query objects for each set.

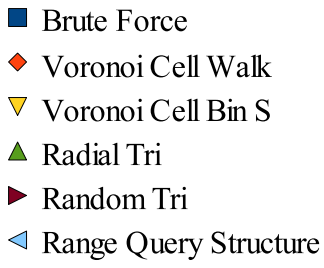

Figure 4: *Legend for Graphs*

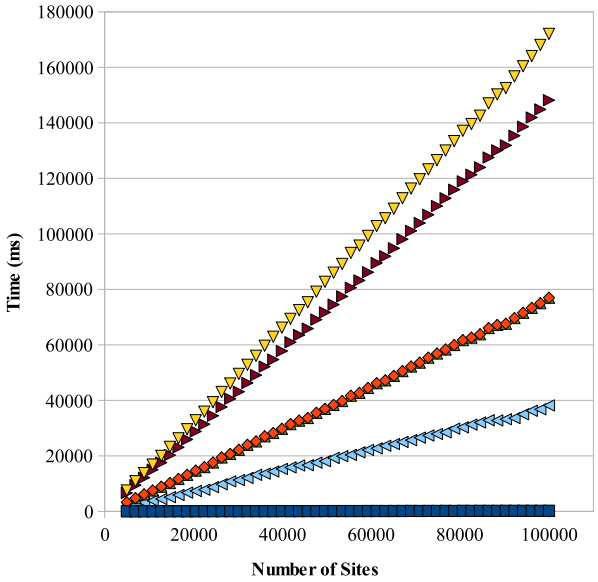

Figure 5: *Initialization Times*

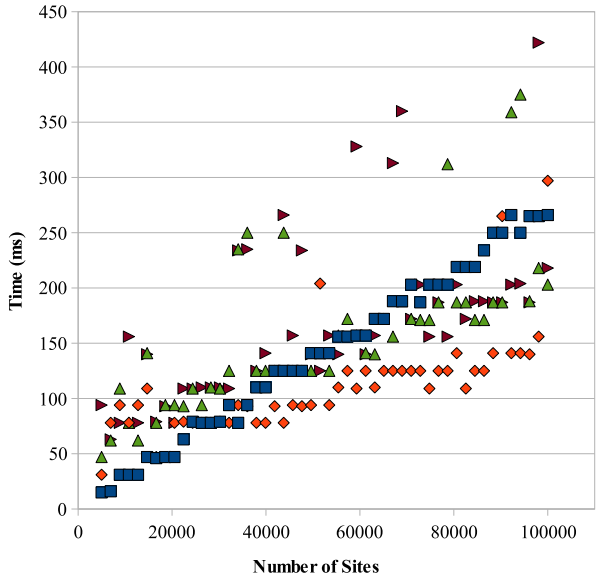

Figure 6: *Query Times for a 0.45 Perimeter Circle*

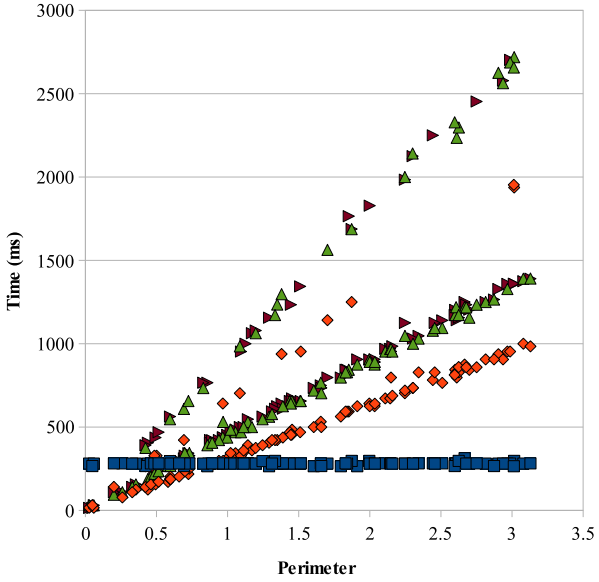

Figure 7: *Query Times for Circles in 100 000 Sites*

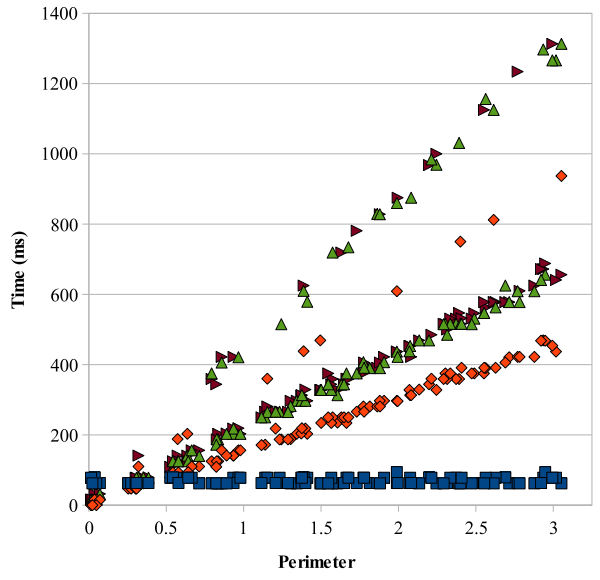

Figure 8: *Query Times for Circles in 25 000 Sites*

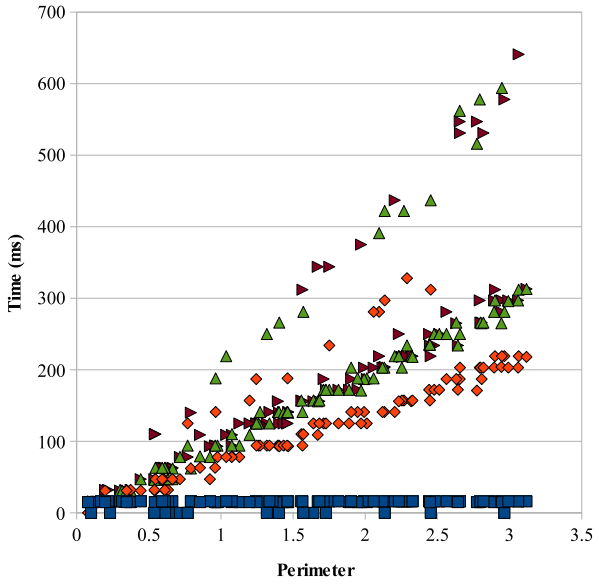

Figure 9: *Query Times for Circles in 5 000 Sites*

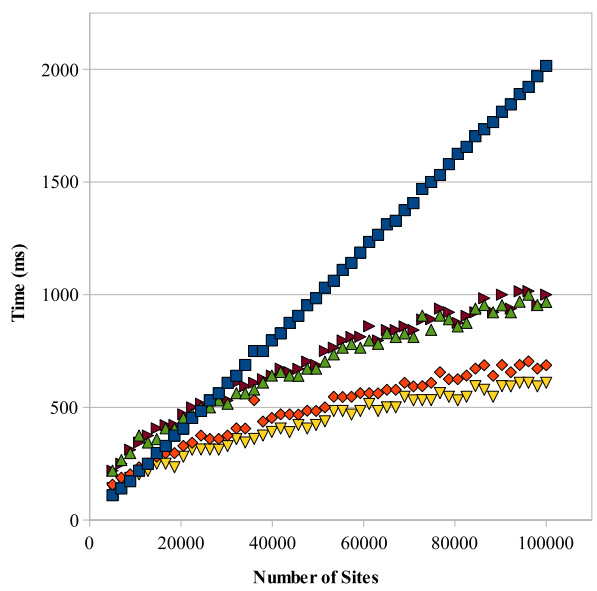

Figure 10: *Query Times for a 1.7 Perimeter Triangle*

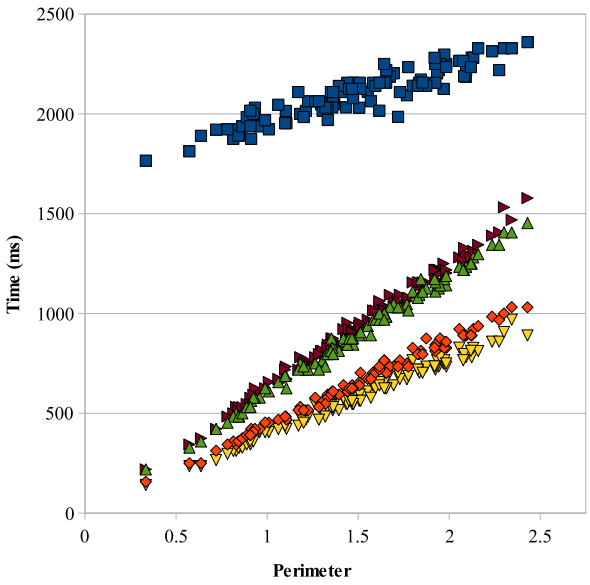

Figure 11: *Query Times for Triangles in 100 000 Sites*

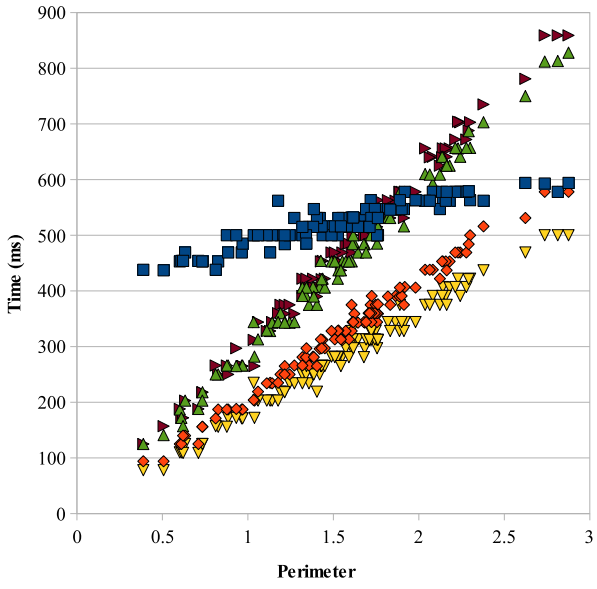

Figure 12: *Query Times for Triangles in 25 000 Sites*

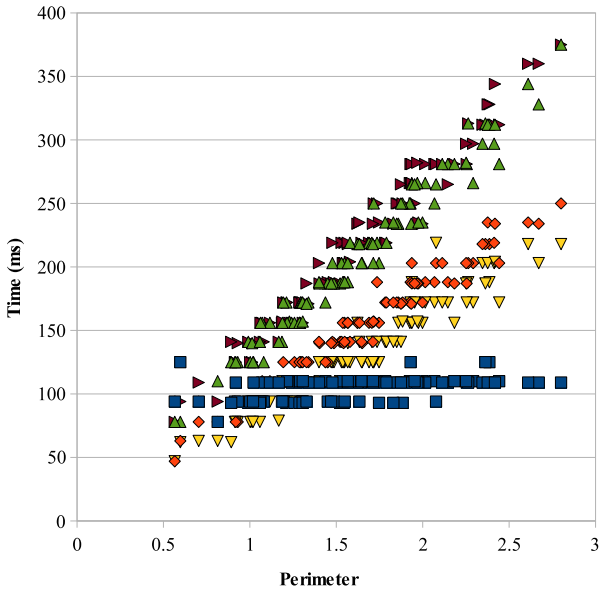

Figure 13: *Query Times for Triangles in 5 000 Sites*

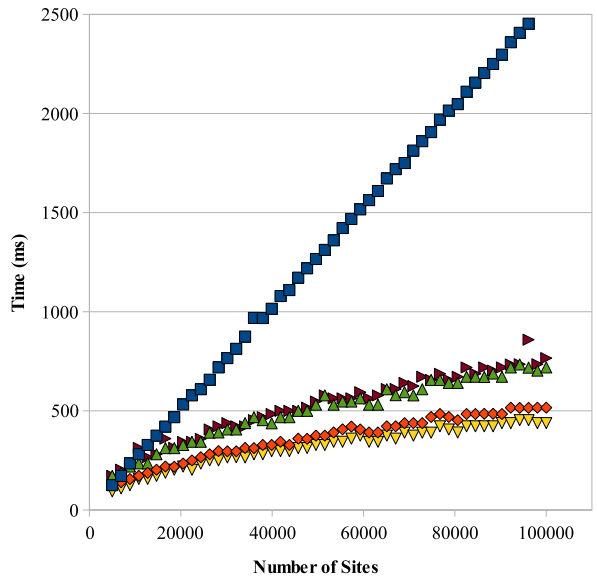

Figure 14: *Query Times for a 1.2 Perimeter Rectangle*

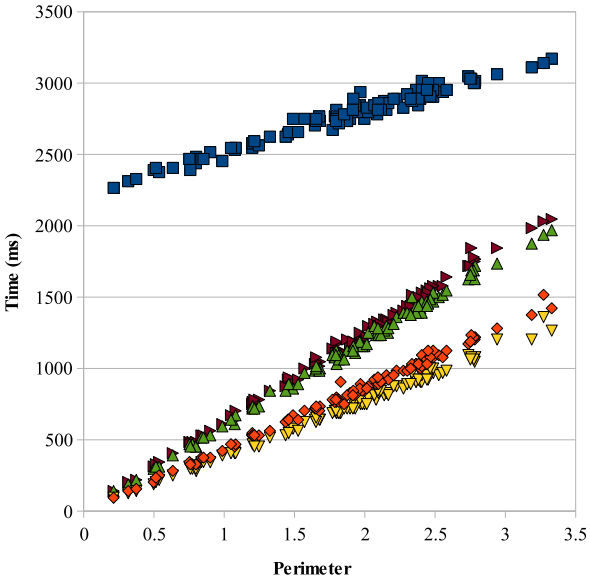

Figure 15: *Query Times for Rectangles in 100 000 Sites*

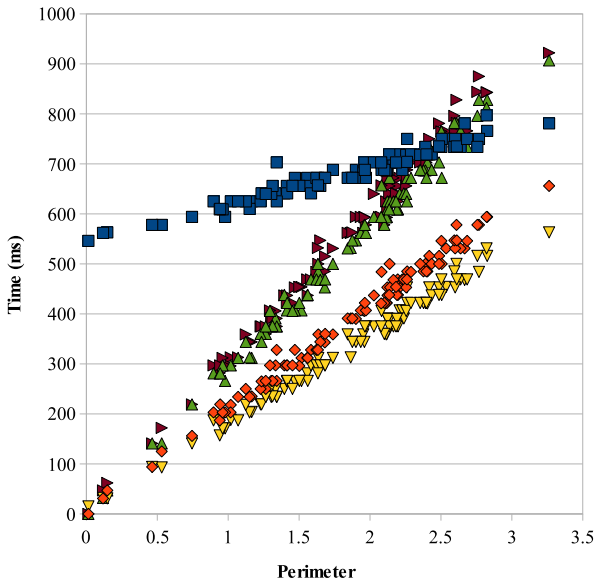

Figure 16: *Query Times for Rectangles in 25 000 Sites*

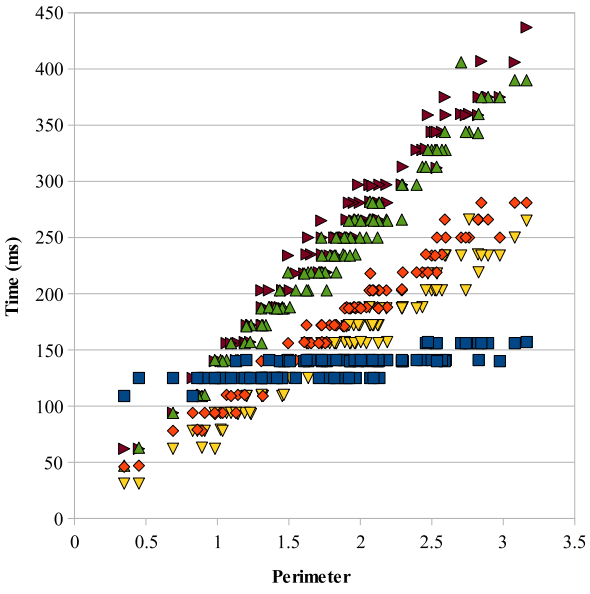

Figure 17: *Query Times for Rectangles in 5 000 Sites*

These graphs show that the Voronoi cell walk and the binary search techniques are advantageous over brute force when the number of sites is large. In addition, query objects with large perimeter are undesirable. Note that in the figures with circle queries, there appear to be two separate groups of data for the non-brute-force techniques. This is due to rounding errors that cause the algorithm to miss the initial point of intersection and travel around the circle a second time. Circle queries do not do so well because the walk-based algorithms rely on (the relatively expensive task of) finding intersections, while the brute force algorithm only relies on (the relatively inexpensive task of) finding distances. Also, the initialization times for the radial triangulation method are the same as those for the Voronoi cell walk.

6 Conclusions

In this paper we have shown that there is an intimate connection between range-queries and closest-point queries for a variety of complex objects. We have also proposed a simple and implementable heuristic that works uniformly for a variety of complex objects. The method works well for large site sets and query objects with small perimeter relative to the site set. This technique can not only be used to answer closest point queries, but also continuous nearest neighbor queries. It is also conceivable to extend this approach for curves in three dimensions.

References

[1] Boost C++ Libraries. http://www.boost.org.

[2] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[3] M. D. Atkinson and J.-R. Sack. Generating binary trees at random. *Inf. Process. Lett.*, 41(1):21–23, 1992.

[4] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.

[5] R. Cole and C. K. Yap. Geometric retrieval problems. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 112–121, 1983.

[6] Victor Teixeira de Almeida. Towards optimal continuous nearest neighbor queries in spatial databases. In *GIS '06: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 227–234, New York, NY, USA, 2006. ACM.

[7] Partha P. Goswami, Sandip Das, and Subhas C. Nandy. Triangle range counting query in 2d and its applications in finding k nearest neighbors of a line segment. *Computational Geometry: Theory and Applications*, pages 163–175, 2004.

[8] D. T. Lee and Y. T. Ching. The power of geometric duality revisited. *Inform. Process. Lett.*, 21:117–122, 1985.

[9] Jiri Matousek. Efficient partition trees. *Discrete Comput. Geom.*, 8(3):315–334, 1992.

[10] P. Mitra. Finding the closest point to a query line. In G. Toussaint, editor, *Snapshots in Computational Geometry*, volume II, pages 53–63. 1992.

[11] P. Mitra and B. B. Chaudhuri. Efficiently computing the closest point to a query line. *Pattern Recognition Letters*, 19:1027–1035, 1998.

[12] Pinaki Mitra, Asish Mukhopadhyay, and S. V. Rao. Computing the closest point to a circle. In *CCCG*, pages 132–135, 2003.

[13] Asish Mukhopadhyay. Using simplicial partitions to determine a closest point to a query line. *Pattern Recognition Letters*, 24:1915–1920, 2003.

[14] S.C. Nandy, S.Das, and P.P.Goswami. An efficient k nearest neighbors searching algorithm for a query line. *TCS: Theoretical Computer Science*, 299(1):273–288, 2003.

[15] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 287–298. VLDB Endowment, 2002.

[16] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[17] E. Welzl. Partition trees for triangle counting and other range searching problems. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 23–33, New York, NY, USA, 1988. ACM Press.