Distance-sensitive point location made easy

Distance-Sensitive Point Location Made Easy*

Boris Aronov[†] Mark de Berg[‡] David Eppstein[§] Marcel Roeloffzen[‡] Bettina Speckmann[‡]

Abstract

Let $\mathcal{S}$ be a planar polygonal subdivision with n edges contained in the unit square. We present a data structure for point location in $\mathcal{S}$ where queries with points far away from any region boundary are answered faster. More precisely, we show that point location queries can be answered in time $O(1 + \min(\log \frac{1}{\Delta_p}, \log n))$, where Δ_p is the Euclidean distance of the query point p to the boundary of the region containing p. Our solution consists of a depth-bounded quadtree and a general point location structure, both of which can be constructed in $O(n \log n)$ time. We also show how to extend the result to convex polyhedral subdivisions in three dimensions.

1 Introduction

Point location is a well-studied problem in computational geometry. Given a subdivision $\mathcal{S}$ the goal is to preprocess it so that we can determine efficiently which region of $\mathcal{S}$ contains a query point p. There are many possible variations of the problem, but we focus on point location in polygonal subdivisions in the plane and convex polyhedral subdivisions in three dimensions.

For planar point location there exist several worst-case optimal solutions. These require $O(n \log n)$ preprocessing, use $O(n)$ space, and can answer a point-location query in $O(\log n)$ time, where n is the number of edges in the subdivision; see the surveys by Preparata [8] and Snoeyink [11] for an overview. In three dimensions no solution is yet known that uses $O(n)$ space and $O(\log n)$ query time. Preparata and Tamassia [9] show that for a convex subdivision one can use a dynamic planar point location structure combined with persistence techniques to obtain an $O(n \log^2 n)$ space structure that allows point location queries in $O(\log^2 n)$ time. Later Snoeyink [11] showed that a similar approach can also be applied to general polyhedral subdivisions in $\mathbb{R}^3$ to obtain an $O(n \log n)$ space structure in $O(n \log n)$ time that allows for point location queries in $O(\log^2 n)$ time.

Although there is a lower bound of $\Omega(\log n)$ on the worst-case query time in planar point location it is possible to improve the query time for certain types of points. This is done for example in entropy-based point location and distance-sensitive point location. For entropy-based point location we have as input a planar subdivision $\mathcal{S}$ with, for each polygon $P_i \in \mathcal{S}$, a probability γ_i that a query falls into P_i. The *entropy* $H(\mathcal{S})$ of such a subdivision is defined as

$$H(\mathcal{S}) := \sum_{R_i \in \mathcal{S}} \gamma_i \log(1/\gamma_i).$$

If the query probabilities are sufficiently skewed, then $H(\mathcal{S}) = o(\log n)$. Iacono [5] showed that an expected query time of $O(H(\mathcal{S}))$ can be achieved with $O(n \log n)$ preprocessing if each region of $\mathcal{S}$ has constant complexity. This result is optimal as the entropy is a lower bound on the expected query time [7, 10]. There have been several other structures that also achieve $O(H(\mathcal{S}))$ expected query time, but are either simpler or have better constants in the query time [2, 3]. The structure presented by Arya, Malamatos, and Mount [2] is relatively simple and efficient in practice. Their solution guarantees that a query point p in polygon P_i takes at most $O(1 + \min(\log(1/\gamma_i), \log n))$ time. Iacono [6] shows that the query distribution does not have to be known in advance and provides a data structure that gradually learns the query distribution and adapts itself, such that it eventually achieves $O(H(\mathcal{S}))$ expected query time. The results mentioned so far all assume that the polygons of the subdivision have constant complexity. To deal with more complex polygons one would first have to decompose each polygon into constant-complexity regions. Collette *et al.* [4] show how to compute a Steiner triangulation of the subdivision $\mathcal{S}$ that has a near-minimal entropy. They also show that the minimal entropy over any Steiner triangulation of $\mathcal{S}$ is a lower bound on the expected query time. Combining this result with the previously mentioned entropy-based point-location structures provides a query structure with near-optimal expected

*B. Aronov has been supported by grant No. 2006/194 from the U.S.-Israel Binational Science Foundation, by NSF Grants CCF-08-30691, CCF-11-17336, and CCF-12-18791, and by NSA MSP Grant H98230-10-1-0210. D. Eppstein has been supported by NSF grant 1217322 and ONR grant N00014-08-1-1015. M. Roeloffzen and B. Speckmann were supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 600.065.120 and 639.023.208, respectively.

[†]Dept. of Computer Science and Engineering, Polytechnic School of Engineering, New York University, USA, aronov@poly.edu

[‡]Dept. of Computer Science, TU Eindhoven, the Netherlands, {mdberg,mroeloff,speckman}@win.tue.nl

[§]Dept. of Computer Science, Donald Bren School of Information and Computer Sciences, University of California, Irvine, eppstein@ics.uci.edu

query time.

In distance-sensitive point location we do not distinguish between queries in different polygons, but instead relate query time to the distance of the query point to the edges of the subdivision. This concept stems from the intuition that if a point is far from the boundary of any subdivision edge, it should be easy to find which region it is in. Moreover, in applications where users are required to select regions one should expect most queries far from the boundary as users are more inclined to click on the 'middle' of a region. Next we define the problem more precisely; we want to answer a query for query point p in $O(1+\min(\log(1/\Delta_p), \log n))$ time, where Δ_p is the minimum Euclidean distance from p to any edge of $\mathcal{S}$. Note that we assume that $\mathcal{S}$ is contained in a square with edge length 1. In a previous paper [1] we showed that this query time can be achieved after $O(n \log n)$ preprocessing time and in $O(n)$ space. That solution relies on creating a refinement of the polygons of $\mathcal{S}$ into a total of $O(n)$ regions such that each region R has constant complexity and the following property: for any point $p \in R$ we have that $\Delta_p = O(\sqrt{\text{area}(R_i)})$. That is, any point p that is far from the boundary of $\mathcal{S}$, must be in a large region R. This refinement is then used as input for an entropy-based point location structure, where we use the area of the regions as the probability distribution. This leads to a solution that has $O(n \log n)$ preprocessing time and answers queries in $O(1 + \min(\log(1/\Delta_p), \log n))$ time in the worst case.

However, the decomposition algorithms is quite complex. It runs in $O(n \log n)$ time, but consists of an elaborate case distinction and requires several other structures to be computed. For example, computing the decomposition requires Voronoi diagrams on the edges of $\mathcal{S}$ in four different distance measures and each of these should be processed for efficient point location. As a result the algorithm would have a large overhead, which would make it difficult to implement and potentially slow in practise. Instead, we propose a much simpler solution, that can also be applied to convex subdivisions in three dimensions.

Our results. We present a distance-sensitive point location structure based on a depth-bounded quadtree and a worst-case optimal point-location structure for planar subdivisions. Both can be computed in $O(n \log n)$ time without requiring many additional or complex structures. Each query for a point p then consists of locating p in the quadtree and potentially finding p in the worst-case optimal point-location structure. Each query takes $O(1 + \min(\log(1/\Delta_p), \log n))$ time. We also show how this approach can be extended to convex polyhedral subdivisions in three dimensions.

2 Connected subdivisions in the plane

The distance-sensitive point location problem has two important requirements. First, any point that is far from the boundary should be located quickly, and second, any point that is close to the boundary should still be located in $O(\log n)$ time. A worst-case optimal point-location structure can be used to satisfy the second requirement and a quadtree where each leaf intersects $O(1)$ features of the subdivision satisfies the first requirement. Unfortunately, neither satisfies both, since the quadtree may have nodes with a very high depth and a worst-case optimal point-location structure gives no guarantees on finding points far from the boundary more quickly. We can however use both structures together to obtain the desired query time.

We construct two structures: a general worst-case optimal point-location structure $\mathcal{PL}(\mathcal{S})$ and a depth-bounded quadtree $\mathcal{QT}(\mathcal{S})$. With a slight abuse of terminology we use *leaf, root* and *node* to denote nodes of the quadtree as well as the square regions they are associated with. The root of the quadtree is the bounding square of $\mathcal{S}$, which we assume to have edge length 1. Each leaf of the quadtree is either empty—it does not intersect the boundary of $\mathcal{S}$—or it has a depth of $\lceil \log \sqrt{n} \rceil$; see Figure 1a. A query for a point p first finds the leaf v that contains p in the quadtree. If v does not intersect any of the boundary elements of $\mathcal{S}$, then the polygon $P \in \mathcal{S}$ that contains v also contains p. If v is not empty, then we conclude that p is close to the boundary of $\mathcal{S}$ and perform a query in $\mathcal{PL}(\mathcal{S})$.

Preprocessing. Constructing a worst-case optimal point-location structure takes $O(n \log n)$ time, where n is the complexity of $\mathcal{S}$. When constructing the quadtree we have to account for the presence of edges of $\mathcal{S}$, and not just its vertices. The standard method to construct a quadtree on a set of points is to recursively split nodes that contain more than one point and propagate the points down the tree such that each leaf stores the points contained in its associated square. In our case each leaf would have to store the edges that intersect it, which would lead to superlinear storage as each edge may intersect many leaves of the quadtree. Instead we use a different approach that uses a sweep-line over the underlying grid of the quadtree.

We first construct the complete quadtree up to depth $\lceil \log \sqrt{n} \rceil$, which represents a grid where each cell has an edge length ℓ between $1/(2\sqrt{n})$ and $1/\sqrt{n}$. It follows that the grid contains $O(n)$ cells in total. We will mark each leaf of the quadtree whose associated grid-cell is intersected by an edge of $\mathcal{S}$. A cell of the grid is intersected by an edge of $\mathcal{S}$ if and only if either one of its boundary segments intersects an edge

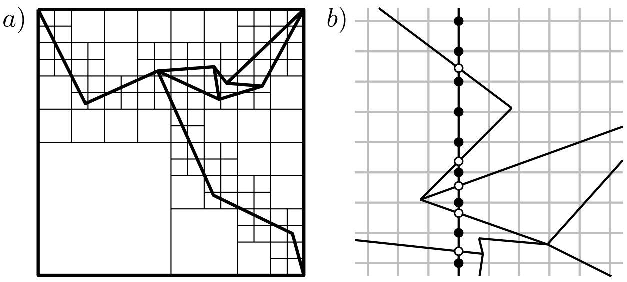

Figure 1: *a*) A depth-bounded quadtree $\mathcal{QT}(\mathcal{S})$. *b*) An illustration of the sweep-line algorithm. Closed disks indicate grid vertices and open disks indicate intersection points of subdivision edges with the sweep-line.

or if the cell contains a vertex of $\mathcal{S}$. We can mark leaves that contain a vertex by locating each vertex within the grid, which takes $O(n \log n)$ time. We then use two sweep-lines to mark cells whose boundary segments are intersected by edges of $\mathcal{S}$. We use a horizontal sweep-line to mark all leaves whose grid cells have their left or right boundary segment intersected by an edge of $\mathcal{S}$. The sweep goes from left to right and we maintain an ordered list of edges from the subdivision that intersect the sweep-line. This ordering changes only when the sweep-line encounters vertices of the subdivision. When the sweep-line encounters a vertex v we locate the vertex in the current edge-ordering in $O(\log n)$ time and then spend $O(k)$ time adding and removing edges adjacent to v, where k is the degree of v. As there are $O(n)$ vertices and the sum of their degrees is $O(n)$ the vertex events take $O(n \log n)$ time in total. When the sweep-line encounters a vertical line of the grid we can start marking cells. Each vertical grid-segment—the boundary edge of one or two cells—is intersected if and only if there are edges of $\mathcal{S}$ between its endpoints on the sweep-line. This is easy to test by simply locating each grid-vertex on the vertical line in the edge-ordering stored in the sweep-line; see Figure 1b. If a grid-segment is intersected by an edge of the subdivision we mark the leaves whose cells are to the left and right of this grid-segment. For each such event we have to perform $O(\sqrt{n})$ binary searches on the edge-ordering of the sweep-line, taking $O(\sqrt{n} \log n)$ time in total. Since there are $O(\sqrt{n})$ such events this takes $O(n \log n)$ time in total. This sweep marks all cells of which the left or right boundary edge is intersected by a subdivision edge. A similar vertical sweep is used to mark all leaves of which the top or bottom segment of its associated grid-cell is intersected by a subdivision edge.

After performing both sweeps each leaf intersected by the subdivision boundary is marked. Next we mark internal nodes of the quadtree of which the associated square intersects an edge of $\mathcal{S}$. We use a bottom-up approach where each node is marked if and only if at least one of its children is marked. Next, the tree is trimmed by removing all nodes with an unmarked parent. The resulting quadtree is a depth-bounded subtree in which each leaf has depth $\lceil \log \sqrt{n} \rceil$ or does not intersect the boundary of the subdivision $\mathcal{S}$. As a final step we do a single point location for each empty leaf of the quadtree to determine which polygon of $\mathcal{S}$ it is contained in and store this information in the leaf.

Lemma 1 *Given a subdivision $\mathcal{S}$, we can construct the depth-bounded quadtree $\mathcal{QT}(\mathcal{S})$ and worst-case optimal point-location structure $\mathcal{PL}(\mathcal{S})$ in $O(n \log n)$ time, where n is the complexity of $\mathcal{S}$.*

Querying. Given the depth-bounded quadtree $\mathcal{QT}(\mathcal{S})$ and the point location structure $\mathcal{PL}(\mathcal{S})$ we perform a point location query on a point p as follows. We first find the leaf v of $\mathcal{QT}(\mathcal{S})$ that contains p. If v is empty, then we report the polygon that contains v, otherwise we do a point location query for p in $\mathcal{PL}(\mathcal{S})$ to find the polygon containing p. Next we show that this indeed provides us with the required query-time.

Lemma 2 *A query as described above for a point p takes $O(\min(1 + \log(1/\Delta_p), \log n))$ time, where Δ_p is the distance from p to the boundary of $\mathcal{S}$.*

Proof. We distinguish two cases. First assume the leaf v from $\mathcal{QT}(\mathcal{S})$ that contains p is empty. Let i denote the depth of v in the quadtree, so we spend $O(i)$ time to locate p. The node v has an edge length of $1/2^i$ and its parent and edge length of $2/2^i$. The parent of v was marked, so it must have intersected the boundary of $\mathcal{S}$. This implies that $\Delta_p \leq 2\sqrt{2}/2^i$, since both p and some point on the boundary of $\mathcal{S}$ are contained in the parent of v. Plugging this in, we find that the query time is

$$\begin{aligned} O(i) &= O(\min(1 + \log(1/(2\sqrt{2}/2^i)), \log n)) \\ &= O(\min(1 + \log(1/\Delta_p), \log n)). \end{aligned}$$

Now suppose v is not empty. In this case we spend $O(\log n)$ time in the quadtree and $O(\log n)$ time in the general point location structure. However, since v must have an edge length of at most $1/\sqrt{n}$ and is intersected by the boundary of $\mathcal{S}$ we know that $\Delta_p \leq \sqrt{2}/\sqrt{n}$ and the query bound follows. $\square$

Combining Lemmas 1 and 2 we obtain the desired result.

Theorem 3 *Given a planar polygonal subdivision $\mathcal{S}$ contained in a square with edge length 1, we can construct in $O(n \log n)$ time and $O(n)$ space a point location structure that can answer a query for a point p in $\mathcal{S}$ in $O(\min(1 + \log(1/\Delta_p), \log n))$ time, where Δ_p*

denotes the distance from p to the boundary of the polygon $P \in \mathcal{S}$ that contains it.

3 Convex subdivisions in $\mathbb{R}^3$

The above method of using a depth-bounded quadtree together with a worst-case optimal point-location structure can also be applied to convex subdivisions in $\mathbb{R}^3$. In this case we would want to compute a depth-bounded octree, where each leaf either does not intersect any boundary facet or has depth $\lceil \log \sqrt[3]{n} \rceil$. As before we can first construct the full octree of depth $\lceil \log \sqrt[3]{n} \rceil$ and then mark leaves that intersect the subdivision boundary. In a general connected subdivision in 3D a cell is intersected if and only if its 2-dimensional faces are intersected by a subdivision facet. The straightforward extension of the sweep-line approach from the 2-dimensional case would require us to maintain a dynamic subdivision defined by the intersection of the input subdivision $\mathcal{S}$ and the sweep-plane. Then whenever the sweep-plane encounters a plane in the grid we should determine if the boundary squares of the grids cells are empty in the sweep-plane. This seems difficult to do in near-linear time. However, in a convex subdivision a grid cell is intersected by a subdivision facet if and only if at least two of its vertices are in different cells of the subdivision. As a result we can simply perform a point location query on each vertex of the grid and test for each grid cell whether all vertices are contained in the same polyhedron of the subdivision. If this is not the case we mark the associated leaf of the octree. We can use the $O(n \log n)$ space structure by Snoeyink [11] to perform each query in $O(\log^2 n)$ time. After marking the leaves of the octree we propagate the marking upwards, trim the tree and determine which regions contain empty leaves as in the two-dimensional case. A query for a point p is again performed by first locating p in the octree, where at most $O(\log n)$ time is spent. If the resulting leaf is not empty we instead find p in the general point location structure in $O(\log^2 n)$ time.

Theorem 4 *Given a 3-dimensional convex polyhedral subdivision $\mathcal{S}$ contained in a cube with edge length 1, we can construct in $O(n \log^2 n)$ time a point location structure that can answer a query for a point p in $\mathcal{S}$ in $O(\log(1/\Delta_p))$ time if $\Delta_p \geq \sqrt{3}/\sqrt[3]{n}$ and $O(\log^2 n)$ otherwise, where Δ_p is the shortest distance from p to nearest boundary facet of $\mathcal{S}$.*

4 Conclusions

We presented a simple data structure for distance-sensitive point location. The data structure consists of a depth-bounded quadtree (or octree) and a data structure for general point location. For a planar subdivision both structures can be constructed in $O(n \log n)$ time and require $O(n)$ space. A query for a point p then takes $O(1 + \min(\log(1/\Delta_p), \log n))$ time, where Δ_p denotes the shortest distance from p to any boundary edge of the subdivision. For a convex subdivision in three dimensions we can construct the octree and general point location structure in $O(n \log^2 n)$ time and $O(n \log n)$ space. A query can then be performed in $O(\log(1/\Delta_p))$ time if $\Delta_p \geq \sqrt{3}/\sqrt[3]{n}$ and $O(\log^2 n)$ otherwise. We believe this method is much simpler than our previous solution as the quadtree is easy to construct and provides very little overhead during querying. An additional advantage is that the method also extends to higher dimensions, although construction of the octree becomes more difficult.

References

[1] B. Aronov, M. de Berg, M. Roeloffzen, and B. Speckmann. Distance-sensitive planar point location. In Proc. *13th Int. Alg. Data Struct. Symp. (WADS)*, pages 49–60, 2013.

[2] S. Arya, T. Malamatos, and D. M. Mount. A simple entropy-based algorithm for planar point location. *ACM Trans. Algorithms*, 3 article 17, 2007.

[3] S. Arya, T. Malamatos, D.M. Mount, and K.C. Wong. Optimal expected-case planar point location. *SIAM J. Comput.*, 37:584–610, 2007.

[4] S. Collette, V. Dujmović, J. Iacono, S. Langerman, and P. Morin. Entropy, triangulation, and point location in planar subdivisions. *ACM Trans. Algorithms*, 8(3):1–18, 2012.

[5] J. Iacono. Expected asymptotically optimal planar point location. *Computational Geometry*, 29(1):19–22, 2004.

[6] J. Iacono and W. Mulzer. A static optimality transformation with applications to planar point location. *Int. J. Comput. Geom. Appl.*, 22(4):327–340, 2012.

[7] D.E. Knuth. *Sorting and Searching, volume 3 of The Art of Computer Programming (2nd edition).* Addison-Wesley, 1998.

[8] F.P. Preparata. Planar point location revisited. *Int. J. Found. Comput. Sci.* 1(1):71–86, 1990.

[9] F.P. Preparata, and R. Tamassia. Efficient point location in convex spatial cell-complex. *SIAM J. Comput.*, 21(2):267-280, 1992.

[10] C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–423, 623–656, 1948.

[11] J. Snoeyink. Point location. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd edition)*, chapter 34. Chapman & Hall/CRC, 2004.