In-Place 2-d Nearest Neighbor Search

Timothy M. Chan Eric Y. Chen
School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
{tmchan, y28chen}@uwaterloo.ca

October 1, 2007

Abstract

We revisit a classic problem in computational geometry: preprocessing a planar n-point set to answer nearest neighbor queries. In SoCG 2004, Brönnimann, Chan, and Chen showed that it is possible to design an efficient data structure that takes no extra space at all other than the input array holding a permutation of the points. The best query time known for such "in-place data structures" is $O(\log^2 n)$. In this paper, we break the $O(\log^2 n)$ barrier by providing a method that answers nearest neighbor queries in time

$$O((\log n)^{\log_{3/2} 2} \log\log n) \;=\; O(\log^{1.71} n).$$

The new method uses divide-and-conquer (based on planar separators) in a way that is quite unlike traditional point location methods, and extends previous 1-d data structuring techniques (specifically the van Emde Boas layout). The method has further applications, for example, in answering extreme point queries for a 3-d point set on the boundary of a convex set of constant complexity.

1 Introduction

The computational geometry perspective. Designing a (static) data structure for n points in the plane so that the nearest neighbor of any query point (in terms of Euclidean distance) can be found quickly is one of the favorite problems in computational geometry, dating back to the early days of the field. Knuth in the 70s dubbed it the "post office problem". A classic result of the field states that with $O(n \log n)$ preprocessing time and $O(n)$ space, nearest neighbor queries can be answered in $O(\log n)$ time. Despite the triviality of the corresponding 1-d problem, this 2-d result requires the development of two fundamental tools [11, 26]—computation of the *Voronoi diagram* (e.g., by divide-and-conquer [30], plane sweep [14], or randomized incremental construction [10]), and the design of *point location* data structures [31] (e.g., by the chain method [13], hierarchical triangulations [23], or persistent search trees [28]).

Despite considerable advances in computational geometry through the years (and recent attention in higher-dimensional approximate nearest neighbor search), some interesting questions remain even for the basic exact nearest neighbor problem in 2-d. For example, among traditional comparison-based algorithms, one could consider refining the constant factors in the number of comparisons performed by a query [29]; or switching to word RAM algorithms for integer input with bounded precision, one could even consider improving the $O(\log n)$ query-time bound itself [9]. The focus of this paper is on another natural question: can one refine the $O(n)$ space bound?

Three years ago, Brönnimann, Chan, and Chen [5] announced a number of new results on space-efficient geometric algorithms and data structures. For 2-d nearest neighbor queries, they showed that surprisingly it is possible to obtain a data structure with zero(!) extra space if we are allowed to permute the array holding the n input points. In the initial version of the paper, the query time is $O(n^\varepsilon)$ for any fixed constant $\varepsilon > 0$; in the full version of the paper, the query time is lowered to $O(\log^2 n)$. The analogous problem in 1-d is obvious, since we can simply sort the given array during preprocessing, without extra storage, and answer queries in $O(\log n)$ time by binary search. However, for the 2-d problem, it is unclear what permutation of the array we should use—obviously sorting by x- or y-order does not work. (For approximate nearest neighbor search in any constant dimension, however, there is a relatively simple solution [8].)

In this paper, we describe a new data structure for 2-d exact nearest neighbor queries that also uses no extra storage other than the input array. Our query time is $O(\log^\gamma n \log\log n)$, where $\gamma = \log_{3/2} 2 < 1.71$.

We believe that the result is interesting, not only because of the fundamental role that the 2-d nearest neighbor problem plays, but also because of the techniques involved. Recall that the problem is equivalent to point location in the Voronoi diagram. Even ignoring the issue that the Voronoi diagram (a planar graph with about $2n$ vertices and $3n$ edges) cannot be stored explicitly, we cannot apply any of the known point location methods directly, since all of them requires $\Omega(n)$ pointers.

Generally, there is no direct way to lay out a tree structure in an array if each element may appear in multiple leaves of the tree; worse, all known linear-space point location structures (the chain method, hierarchical triangulations, persistent search trees, ...) are not trees but dags (see [29] for why). Indirectly, however, one can encode a bit "within" $O(1)$ elements and thus a pointer within $O(\log n)$ elements by a well-known trick (see Section 4.3), but each pointer access would then require logarithmic cost. Indeed, the previous $O(\log^2 n)$ method [5] was obtained this way: we first reduce the size of the planar graph under consideration by a logarithmic factor (using graph separators); we then encode an entire point location structure within the array, where a normal $O(\log n)$-time query algorithm is simulated with a slow-down of a logarithmic factor.

To beat $O(\log^2 n)$, we have to abandon the idea of simulating an existing point location algorithm, and instead work from scratch. Indeed, we propose a brand new algorithm for point location in the Voronoi diagram, by using planar graph separators in a recursive fashion. Although this is hardly the first time separators are used in computational geometry (or in point location algorithms for that matter), the way recursion is used is different and, in our opinion, quite fascinating (as one might suspect from the relative unusualness of our time bound).

The data structure perspective. We believe our result is of interest not just to computational geometers but to the broader data structures community. "In-place data structures" that use no extra space beyond the input array (also called *implicit data structures*) have had a long history; the standard binary heap is one of the earliest and most familiar examples.

In-place data structures for the 1-d successor search problem in the dynamic case, with insertions and deletions, have been the subject of intense study: after some early attempts, Munro [25] in FOCS'84 obtained the first polylogarithmic method with $O(\log^2 n)$ update and query time. His method is based on modifying a balanced search tree using the above-mentioned pointer-encoding scheme with logarithmic-factor slow-down. At some point, it was conjectured that $O(\log^2 n)$ is optimal. It took 16 years for researchers to break the $O(\log^2 n)$ barrier in this instance: Franceschini, Grossi et al. [18, 16, 17] (FOCS'02, SODA'03, and WADS'03) improved the time per operation to $O(\log^2 n/\log\log n)$, then to $O(\log n \log\log n)$, and eventually to $O(\log n)$.

Static 2-d nearest neighbor search is arguably the most natural extension of 1-d successor search and the logical "next step" to consider. The fundamental issue is no longer how to encode a tree in an array, but more challengingly, how to encode a planar graph (the Voronoi diagram). Although our result is stated in terms of nearest neighbor search, our approach can potentially handle point location in embeddings of other canonically defined planar graphs (e.g., the 2-d projection of certain lower envelopes of planes).

There has been previous work on space-efficient representations of planar graphs, specifically, triangulations. For instance, Aleardi et al. [1, 2] gave *succinct data structures* for answering navigational queries in a triangulation (point location queries are more powerful than navigational queries, though to be fair, the authors were aiming for constant query time). The space requirement of succinct data structures typically approaches the information-theoretic minimum encoding length, which in the case of triangulations is $O(n)$ bits, i.e., $O(n/\log n)$ words only. However, in the geometric setting with coordinates attached to data elements, Aleardi et al.'s structures still require the data elements (triangles) to be stored in a specific permutation in the input array, in addition to the $O(n)$-bit index. These structures are also obtained by less powerful graph partitioning strategies (the separator theorem is not even required).

The techniques underlying our new method bear striking connections with known data structuring techniques: our organization of the input array can be viewed as a more sophisticated extension of the "van Emde Boas layout" [12, 27, 33], popular in the context of cache-oblivious data structures. We perform n^β-way divide-and-conquer, for some constant β, to ensure that the logarithm of the input size—hence, the size of pointers—decreases exponentially, thus mitigating the effect of the logarithmic-factor slow-down.

We remark that due to not only theoretical but also practical interest, in-place algorithms and data structures are currently undergoing a revival, as emerging applications involving massive data sets have brought space consideration to the forefront. Very recent work has appeared on traditional sorting problems (e.g., [15, 20]), computational geometry (e.g., [3, 4, 6, 32]), and string problems (e.g., [19]). (We should mention that while our data structure does not require extra space, our preprocessing algorithm is regrettably not an in-place algorithm.)

Our paper has a simple organization. After some preliminaries in the next section, the data structure and query algorithm are described in Section 3 (with three interdependent subroutines provided in three subsections), which are then analyzed in Section 4. We conclude with remarks and open problems in the last section.

2 Preliminaries

2.1 The Permutation+Bits Model

To simplify the presentation of our in-place data structure, we will work mostly in an intermediate model. Here, in addition to the input array holding a permutation of the given points, the data structure may also have an extra array of bits. The content of this extra space can only be accessed through bit probes, with each bit access costing unit time. For lack of a better name, we call this the *permutation+bits* model.

Curiously, even if we are allowed to use a huge number of extra bits (quadratic or worse), the problem remains nontrivial in this model. For example, consider one of the most naive solutions to point location: the *slab method* [26]. We draw vertical lines at every vertex of the planar subdivision (the Voronoi diagram), and store the line segments of the subdivision at each of the $O(n)$ vertical slabs in sorted y-order. To answer a query, we first perform a binary search in x

to find the slab containing the query point q, and perform another binary search in y to find segment immediately above q. This method uses near-quadratic space but has asymptotically the best query time possible, $O(\log n)$, in the standard model. However, to support binary search, we need to store each of the $O(n)$ sorted lists in subarrays. In the permutation+bits model, one list can be represented within the input array itself, but with multiple lists sharing common elements, in general one needs to tap into the extra array of bits. Each occurrence of a point beyond its first can only be represented implicitly through pointers. (We assume that points are indivisible data types—think of them as infinite-precision real numbers.) With the bit probe restriction, each access to a point would require logarithmic cost, so the binary search would now cost $O(\log^2 n)$!

Nevertheless, by storing the point set in a carefully arranged order, we show how to beat the $O(\log^2 n)$ bound, and at the same time keep the number of extra bits linear (at most δn). This is sufficient to yield our final in-place result, as a well-known trick can reduce the amount of extra space from linear to zero (as we eventually reveal in Section 4.3).

2.2 A Permutation from Voronoi Diagrams and Separators

We now specify the order for the input array. For this, we apply a standard, multiple-clusters version of the planar separator theorem [24] to the dual of the Voronoi diagram, where the removal of a set of separator Voronoi cells produces $O(b)$ clusters of $O(n/b)$ Voronoi cells. Let $\mathrm{Vor}(P)$ denote the Voronoi diagram of a point set P and $\mathrm{Vor}(p, P)$ denote the Voronoi cell (a convex polygon) of the point $p \in P$ in $\mathrm{Vor}(P)$. The theorem implies:

Lemma 2.1 (Separator Theorem) *Given a set P of n points in the plane and a parameter $1 \leq b \leq n$, we can partition P into a subset P_S (the* separator*) of $O(\sqrt{bn})$ points and $O(b)$ subsets $P_1, P_2 \ldots$ (the* clusters*) each of $O(n/b)$ points, so that:*

For every two points $p, p' \in P - P_S$, if $\mathrm{Vor}(p, P)$ is adjacent to $\mathrm{Vor}(p', P)$, then p and p' belong to the same cluster.

To generate the permutation for P, we simply generate the permutations for the subsets P_S, P_1, P_2,... recursively and concatenate these permutations. The choice of the parameter b will be specified later.

Applying this recursive procedure to the input point set P_0, we implicitly obtain a tree T of subsets, where the root holds the global set P_0, and each generated subset P resides in a contiguous subarray of the input array. It is important to note that we are working not with one global Voronoi diagram but with Voronoi diagrams of subsets. Understanding the relationship between Voronoi diagrams of different subsets will be vital (see Observations 3.1 and 3.2 to come). We will make use of the fortunate fact that nearest neighbor search is a "decomposable" problem—the nearest neighbor of a point in a union of two subsets can be obtained from the nearest neighbor in each of the two subsets trivially.

Once the permutation is fixed, our subsequent data structures cannot change the order of the input array but can only append to the extra array of bits. Note the resemblance to the van Emde Boas layout [12] (P_S is the "top" structure and $P_1, P_2, \ldots$ are the "bottom" structures—in a query, the top structure helps us determine which bottom structure to recurse in).

In this and the next section, we use N to denote the size of the global point set P_0, and reserve the symbol n for the size of an arbitrary subset P in the tree T.

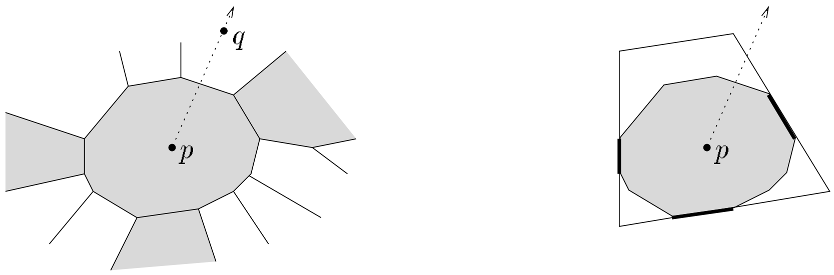

Figure 1: (Left) A cell $\mathrm{Vor}(p, P)$ ($p \in P_S$) drawn with a ray $\overrightarrow{pq}$; the separator cells are darkened. (Right) The same cell $\mathrm{Vor}(p, P)$ drawn inside the larger cell $\mathrm{Vor}(p, P_S)$.

3 The Query Algorithm

3.1 Nearest Neighbor Queries

With the tree structure T defined in Section 2.2, the following query algorithm is natural to think of. To find the nearest neighbor of q in a given subset P in T, we first find the nearest neighbor p of q in the separator subset P_S recursively. If the actual nearest neighbor is not in P_S, we deduce which cluster P_i it belongs to and make a second recursive call to find the nearest neighbor of q in P_i.

To determine the right cluster to recurse in, we find the edge of $\mathrm{Vor}(p, P)$ hit by the ray $\overrightarrow{pq}$ and let P_i be the cluster containing the point in $P - \{p\}$ that defines this edge. See Figure 1 (left). Finding this edge is simply a "one-dimensional" problem along the boundary of the convex polygon $\mathrm{Vor}(p, P)$ and is doable by binary search.

To see correctness, observe that all points on the line segment $\overline{pq}$ have the same nearest neighbor in P_S, namely p, by convexity of $\mathrm{Vor}(p, P_S)$. Thus, all cells $\mathrm{Vor}(p', P)$ hit by $\overline{pq}$, with the exception of $\mathrm{Vor}(p, P)$, must have $p' \notin P_S$. So, all such points p', including the nearest neighbor of q in P, must belong to the same cluster.

Unfortunately, the above approach has one major flaw: the binary search is trivially implementable in the traditional model, but not so in the permutation+bits model. To form the one-dimensional lists for all the convex polygons $\mathrm{Vor}(p, P)$, we need pointers with logarithmic number of bits, but this would slow down search to $O(\log^2 n)$ time, for the same reason mentioned in Section 2.1. (If all convex polygons have constant size, this issue goes away, but there could be many points with many Voronoi neighbors.) Secondly, the number of extra bits used would be slightly superlinear.

At least the second issue can be resolved, by the following observation: to find the cluster P_i, it suffices to search in $\mathrm{Vor}(p, P_S)$ instead of $\mathrm{Vor}(p, P)$. This observation is helpful, as $|P_S|$ is much smaller than $|P|$ (though not small enough to make the problem trivial, as $|P_S| \gg \sqrt{|P|}$ in the separator theorem). Throughout the paper, the $\widetilde{O}$ notation hides polylogarithmic factor.

Observation 3.1 *We can preprocess a subset P in T into a data structure with $\widetilde{O}(|P_S|)$ bits so that:*

Given a query ray $\overrightarrow{pq}$ with $p \in P_S$ and given the point $p' \in P_S - \{p\}$ that defines the edge e of $\mathrm{Vor}(p, P_S)$ hit by $\overrightarrow{pq}$, we can determine the cluster P_i containing the point in $P - \{p\}$ that defines the edge of $\mathrm{Vor}(p, P)$ hit by $\overrightarrow{pq}$ in $O(\log |P|)$ time.

Proof Consider the boundary of the convex polygon $\mathrm{Vor}(p, P)$. Blacken those edges that are adjacent to separator cells $\mathrm{Vor}(p', P)$ with $p' \in P_S - \{p\}$; color the remaining subchains white.

Since the points in $P - \{p\}$ that define edges in the same white subchain lie in the same cluster, it suffices to determine which white subchain is hit by the query ray.

Consider the relationship between the two convex polygons $\mathrm{Vor}(p, P_S)$ and $\mathrm{Vor}(p, P)$. The former contains the latter. Furthermore, the black edges of $\mathrm{Vor}(p, P)$ remain on the Voronoi diagram of P_S and thus lie on the boundary of $\mathrm{Vor}(p, P_S)$. See Figure 1 (right). Determining which white subchain of $\mathrm{Vor}(p, P)$ is hit by a ray reduces to determining which *cap*—i.e., connected component of $\mathrm{Vor}(p, P_S) - \mathrm{Vor}(p, P)$—is hit by the ray.

If we know the edge e of $\mathrm{Vor}(p, P_S)$ hit by the ray, there are only one or two caps that involve e and that may be hit by the ray. In the latter case, we can deduce which of the two is the answer by comparing the ray with an arbitrary vertex v on $e \cap \mathrm{Vor}(p, P)$.

During preprocessing, we simply precompute all answers in a table: for each pair p and p' defining a Voronoi edge e of $\mathrm{Vor}(P_S)$, we store the cluster index for each of the at most two caps, and also the vertex v if necessary. Since the number of edges in a 2-d Voronoi diagram is linear, the table can be stored within the array of extra bits of size $\widetilde{O}(|P_S|)$ by using perfect hashing [21]; the parameters associated with the required hash functions can be stored as well. The normal $O(1)$-time worst-case query algorithm for perfect hashing now takes $O(\log |P|)$ time, since each entry of the table (and parameter of the hash functions) requires logarithmic number of bits. □

To summarize, the main bottleneck is the following subproblem:

> *The Ray Problem.* Preprocess a subset P in the tree T so that given a query ray $\overrightarrow{pq}$ originating from a point $p \in P$, we can quickly find the point $p' \in P - \{p\}$ that defines the edge of $\mathrm{Vor}(p, P)$ hit by $\overrightarrow{pq}$. Equivalently, we want to find the first line $\ell(p, p')$ hit by $\overrightarrow{pq}$ over all $p' \in P - \{p\}$, where $\ell(p, p')$ denotes the bisector between p and p'.

If there is an efficient solution to this ray problem, called Ray-Query(P, $\overrightarrow{pq}$), then we can answer nearest neighbor queries by the following pseudocode NN-Query() (omitting obvious base cases).

Algorithm NN-Query(P, q):
1. $p =$ NN-Query(P_S, q)
2. $p' =$ Ray-Query(P_S, $\overrightarrow{pq}$)
3. determine i from p and p' by Observation 3.1
4. $p'' =$ NN-Query(P_i, q)
5. if q is closer to p'' than p then return p'' else return p

A remark on implementation: We assume that the sizes of P_S and the P_i's have been stored as part of the data structure (which requires only $\widetilde{O}(b)$ extra space, which is smaller than $|P_S| = O(\sqrt{bn})$). A point is represented by its location in the current input subarray. We have ignored issues surrounding standard address calculations in both the input array and the extra bit array, which are doable via offsets. These calculations are essential to ensure $\log n$ pointer costs indeed decrease as the number of points n in the subarray decreases.

Let $S(n)$ and $S_{\mathrm{ray}}(n)$ denote the number of bits required by a data structure for the nearest neighbor and the ray problem for $|P| = n$ respectively. Let $Q(n)$ and $Q_{\mathrm{ray}}(n)$ denote the query time for these two problems. Then we have the recurrences:

$$S(n) = S(n_S) + \sum_i S(n_i) + S_{\mathrm{ray}}(n_S) + \widetilde{O}(n_S) \qquad (1)$$

$$Q(n) = Q(n_S) + \max_i Q(n_i) + Q_{\mathrm{ray}}(n_S) + O(\log n), \qquad (2)$$

where $n_S = |P_S| = O(\sqrt{bn})$, $n_i = |P_i| = O(n/b)$, and $n_S + \sum_i n_i = n$.

3.2 Ray Queries

Although the ray problem appears easier (more "one-dimensional") than the original problem, for some time we were not able to come up with an efficient algorithm under the permutation+bits model, until we gave up on the binary search approach altogether. Our new idea is simple: why not exploit the 2-d tree structure T that we already have to solve this one-dimensional problem as well? The query time will not be logarithmic (but will be $o(\log^2 n)$), and we will need significantly more than linear space, but fortunately the ray problem is only required for small (separator) subsets anyway (as we have used Observation 3.1).

Again we are unable to solve the problem directly but need a subroutine for another subproblem, a bichromatic version of the ray subproblem. This version has two main differences: the source of a ray is not from the same given subset P but is from another subset Q, and the Voronoi diagram of interest is not $\mathrm{Vor}(P)$ but $\mathrm{Vor}(P \cup \{p\})$ for a query point $p \in Q$.

> *The Bichromatic Ray Problem.* Preprocess two disjoint subsets P (the "red" points) and Q (the "blue" points) in the tree T so that given a query ray $\overrightarrow{pq}$ originating from a point $p \in Q$, we can quickly find the point $p' \in P$ that defines the edges of $\mathrm{Vor}(p, P \cup \{p\})$ hit by $\overrightarrow{pq}$. Equivalently, we want to find the first bisector line $\ell(p, p')$ hit by $\overrightarrow{pq}$ over all $p' \in P$.

If there is an efficient solution to the above subproblem, called Bichromatic-Ray-Query(P, Q, $\overrightarrow{pq}$), then we can solve the ray problem by the following pseudocode:

Algorithm Ray-Query(P, $\overrightarrow{pq}$), where $p \in P$:

1. if $p \in P_S$ then
2. $\quad$ $p' =$ Ray-Query(P_S, $\overrightarrow{pq}$)
3. $\quad$ determine i from p and p' by Observation 3.1
4. $\quad$ $p'' =$ Bichromatic-Ray-Query(P_i, P_S, $\overrightarrow{pq}$)
5. else
6. $\quad$ determine i with $p \in P_i$
7. $\quad$ $p' =$ Bichromatic-Ray-Query(P_S, P_i, $\overrightarrow{pq}$)
8. $\quad$ $p'' =$ Ray-Query(P_i, $\overrightarrow{pq}$)
9. if $\overrightarrow{pq}$ hits $\ell(p, p'')$ before $\ell(p, p')$ then return p'' else return p'

The correctness immediately follows from the same Observation 3.1 and the "decomposability" of the ray problem. Note that line 6 takes $O(1)$ time.

Let $S_{\mathrm{bi}}(n, m)$ and $Q_{\mathrm{bi}}(n, m)$ denote the number of bits and query time required by a data structure for the bichromatic ray problem for $|P| = n$ and $|Q| = m$ respectively. Then we have the recurrences:

$$S_{\mathrm{ray}}(n) \;=\; S_{\mathrm{ray}}(n_S) + \sum_i S_{\mathrm{ray}}(n_i) + \sum_i (S_{\mathrm{bi}}(n_i, n_S) + S_{\mathrm{bi}}(n_S, n_i)) + \widetilde{O}(n_S) \tag{3}$$

$$Q_{\mathrm{ray}}(n) \;=\; \max\left\{ Q_{\mathrm{ray}}(n_S) + \max_i Q_{\mathrm{bi}}(n_i, n_S),\ \max_i (Q_{\mathrm{ray}}(n_i) + Q_{\mathrm{bi}}(n_S, n_i)) \right\} + O(\log n). \tag{4}$$

3.3 Bichromatic Ray Queries

To complete the solution, we need to present an algorithm for the bichromatic ray problem. For this, we use the same recursive approach for a third (and final!) time. This time, we need a variant of Observation 3.1 given below, where the space usage might at first appear too large but turns out not to be a problem if parameters are chosen carefully enough.

Observation 3.2 *We can preprocess two subsets P and Q of T into a data structure with $\widetilde{O}(|P_S| \cdot |Q|)$ bits so that:*

Given a ray $\overrightarrow{pq}$ with $p \in Q$ and a point $p' \in P_S$ that defines the edge of $\mathrm{Vor}(p, P_S \cup \{p\})$ *hit by $\overrightarrow{pq}$, we can determine the index i to the cluster for the point in P that defines the edge of* $\mathrm{Vor}(p, P \cup \{p\})$ *hit by $\overrightarrow{pq}$ in $O(\log |P|)$ time.*

Proof We modify the proof of Observation 3.1, considering $\mathrm{Vor}(p, P \cup \{p\})$ instead of $\mathrm{Vor}(p, P)$. We blacken those edges that are adjacent to $\mathrm{Vor}(p', P \cup \{p\})$ with $p' \in P_S$. We add one easy fact: for any two points $p_1, p_2 \in P$, if $\mathrm{Vor}(p_1, P \cup \{p\})$ and $\mathrm{Vor}(p_2, P \cup \{p\})$ are adjacent, then $\mathrm{Vor}(p_1, P)$ and $\mathrm{Vor}(p_2, P)$ are also adjacent. Thus, points in P that define edges in the same white subchain still lie in the same cluster.

By the same reasoning as before, for each pair $p \in Q$ and $p' \in P_S$, we can generate at most two possible answers. We store them all in a table as before, except that this time we don't need hashing. The size of the two-dimensional table is trivially $\widetilde{O}(|P_S| \cdot |Q|)$. Note that the query time $O(\log |P|)$ does not depend on Q, since each entry of the table has $O(\log |P|)$ bits. □

The pseudocode for bichromatic ray queries is as follows:

Algorithm Bichromatic-Ray-Query(P, Q, $\overrightarrow{pq}$), where $p \in Q$:
1. p' = Bichromatic-Ray-Query(P_S, Q, $\overrightarrow{pq}$)
2. determine i from p and p' by Observation 3.2
3. p'' = Bichromatic-Ray-Query(P_i, Q, $\overrightarrow{pq}$)
4. if $\overrightarrow{pq}$ hits $\ell(p, p'')$ before $\ell(p, p')$ then return p'' else return p'

Correctness is self-evident. We have these recurrences:

$$S_{\mathrm{bi}}(n, m) = S_{\mathrm{bi}}(n_S, m) + \sum_i S_{\mathrm{bi}}(n_i, m) + \widetilde{O}(n_S m) \tag{5}$$

$$Q_{\mathrm{bi}}(n, m) = Q_{\mathrm{bi}}(n_S, m) + Q_{\mathrm{bi}}(n_i, m) + O(\log n). \tag{6}$$

4 Analysis

4.1 First Version

For the analysis, we first describe a simple choice of b that leads to an $o(\log^2 n)$ query time bound with linear space: namely, we set $b = n^{0.22}$ for each subset P of size n in the tree T. Here, $n_S = O(\sqrt{bn}) = O(n^{0.61})$ and $n_i = O(n/b) = O(n^{0.78})$. The solutions to the recurrences, while technically straightforward, are delicate and rely on two numerical facts: $0.61^{1.95} + 0.78^{1.95} < 1$ and

$0.61 \cdot 1.61 < 1$. Below, c is some constant and $\varepsilon > 0$ is any sufficiently small constant.

$$\begin{array}{llcll}
\text{by (6):} & Q_{\text{bi}}(n,m) & = & Q_{\text{bi}}(cn^{0.61},m) + Q_{\text{bi}}(cn^{0.78},m) + O(\log n) & \\
 & & & & \Longrightarrow\ Q_{\text{bi}}(n,m) \ = \ O(\log^{1.95-\varepsilon} n) \\
\text{by (4):} & Q_{\text{ray}}(n) & = & Q_{\text{ray}}(cn^{0.78}) + O(\log^{1.95-\varepsilon} n) & \Longrightarrow\ Q_{\text{ray}}(n) \ = \ O(\log^{1.95} n) \\
\text{by (2):} & Q(n) & = & Q(cn^{0.61}) + Q(cn^{0.78}) + O(\log^{1.95-\varepsilon} n) & \Longrightarrow\ Q(n) \ = \ O(\log^{1.95} n) \\
\text{by (5):} & S_{\text{bi}}(n,m) & = & \widetilde{O}(nm) & \\
\text{by (3):} & S_{\text{ray}}(n) & = & \widetilde{O}(n^{1.61}) & \\
\text{by (1):} & S(n) & = & S(n_S) + \sum_i S(n_i) + \widetilde{O}((n^{0.61})^{1.61}) & \Longrightarrow\ S(n) \ = \ O(n).
\end{array}$$

4.2 Second Version

Now, to improve upon the $O(\log^{1.95} n)$ query bound, we use a more clever choice of b, switching between two cases. We first mark subsets in the tree T as *ordinary* or *special* according to the following rules:

> The global set P_0 is ordinary. If P is ordinary, then P_S is special but $P_1, P_2, \ldots$ are ordinary. If P is special, then $P_S, P_1, P_2, \ldots$ are all special.

For each special subset P of size n, we set $b = n^{1/3}$, so that $n_S = O(\sqrt{bn}) = O(n^{2/3})$ and $n_i = O(n/b) = O(n^{2/3})$. For each ordinary subset P of size n, we set $b = n^{2\varepsilon}$, so that $n_S = O(n^{1/2+\varepsilon})$ and $n_i = O(n^{1-2\varepsilon})$.

The solution to the recurrences becomes as follows, where $\gamma := \log_{3/2} 2 < 1.71$, and functions $Q(\cdot), S(\cdot), \ldots$ marked with superscripts * represent the specified time/space requirement for special subsets P, while functions without superscripts are for ordinary subsets P:

$$\begin{array}{llcll}
\text{by (6):} & Q^*_{\text{bi}}(n,m) & = & 2Q^*_{\text{bi}}(cn^{2/3},m) + O(\log n) & \Longrightarrow\ Q^*_{\text{bi}}(n,m) \ = \ O(\log^{\gamma} n) \\
\text{by (4):} & Q^*_{\text{ray}}(n) & = & Q^*_{\text{ray}}(cn^{2/3}) + O(\log^{\gamma} n) & \Longrightarrow\ Q^*_{\text{ray}}(n) \ = \ O(\log^{\gamma} n) \\
\text{by (2):} & Q^*(n) & = & 2Q^*(cn^{2/3}) + O(\log^{\gamma} n) & \Longrightarrow\ Q^*(n) \ = \ O(\log^{\gamma} n \log\log n) \\
 & Q(n) & = & Q^*(cn^{1/2+\varepsilon}) + Q(cn^{1-2\varepsilon}) + O(\log^{\gamma} n) & \\
 & & = & Q(cn^{1-2\varepsilon}) + O(\log^{\gamma} n \log\log n) & \Longrightarrow\ Q(n) \ = \ O(\log^{\gamma} n \log\log n) \\
\text{by (5):} & S^*_{\text{bi}}(n,m) & = & \widetilde{O}(nm) & \\
\text{by (3):} & S^*_{\text{ray}}(n) & = & \widetilde{O}(n^{5/3}) & \\
\text{by (1):} & S^*(n) & = & \widetilde{O}((n^{2/3})^{5/3}) \ = \ \widetilde{O}(n^{10/9}) & \\
 & S(n) & = & S^*(cn^{1/2+\varepsilon}) + \sum_i S(n_i) + \widetilde{O}((n^{1/2+\varepsilon})^{5/3}) & \\
 & & = & \sum_i S(n_i) + o(n^{1-\varepsilon}) & \Longrightarrow\ S(n) \ = \ O(n).
\end{array}$$

We can in fact make the space bound at most δN for an arbitrarily small constant $\delta > 0$: just terminate the recursion when $|P| = n$ drops below a sufficiently large constant. The solution to the $S(n)$ recurrence now yields at most δn. All three types of queries can be handled trivially in $O(1)$ time in the base cases when the subset P has constant size, and so the overall query time is still $O(\log^{\gamma} N \log\log N)$.

4.3 In-Place Version

Finally, we convert our data structure in the permutation+bits model into an in-place data structure with no extra space at all.

We apply the following well-known bit-encoding trick, used in many previous work on in-place algorithms (e.g., [25]): Divide the input array into consecutive pairs. For each pair (p, q), we compare p and q lexicographically, and permute the pair in the array to encode a bit so that if the bit is 0, the smaller point appears first, else the larger point appears first. (We may assume no two points are identical, by removing duplicates.) As long as $\delta < 1/2$, this scheme can encode the entire data structure within the array.

We argue that our data structure remains valid after such pairs are permuted. First, in the application of the separator theorem, assuming the given set has even size, we may ensure that all subsets have even size. Indeed, if a cluster P_i has odd size, we can move a point over to P_S; the separator size $|P_S|$ would still be $O(\sqrt{bn} + b) = O(\sqrt{bn})$. Thus, subsets in our tree T still reside in contiguous input subarrays. When the data structure refers to a point p, the actual location i of p may now be off by 1. We use a different representation of p that is unaffected by permutations of pairs: namely, we take the index $\lfloor i/2 \rfloor$ together with an extra bit indicating whether p is the smaller or the larger point of the $\lfloor i/2 \rfloor$-th pair in the input array. Given the index and the bit, we can recover the actual location of p.

We conclude:

Theorem 4.1 *Given an array of N points in the plane, we can permute the array without using extra storage so that given any query point, its nearest neighbor can be found in $O(\log^{1.71} N)$ time.*

5 Final Remarks

We have omitted preprocessing cost in the analysis, but it is easy to see that our data structure can be built in $O(n \log n)$ deterministic time by a non-in-place algorithm (using a known efficient deterministic hashing scheme such as [22]).

Our algorithm uses only limited properties about Euclidean Voronoi diagrams and so is likely generalizable to solve other problems. For instance, by the same approach, we can answer point location queries in the xy-projection of the 3-d lower envelope of n planes tangent to a given convex algebraic surface of constant degree. The 2-d Euclidean nearest neighbor problem corresponds to the special case where the surface is the unit paraboloid [11]. In the dual, we can answer answer extreme point queries (finding the minimal point along a given direction) for a 3-d lower hull where all vertices lie on a convex surface of constant complexity. It is unclear at the moment how to obtain $o(\log^2 n)$ query time for certain related problems, such as point location in the 2-d Delaunay triangulation (or projection of a 3-d lower hull), as these problems are not directly decomposable.

The most obvious open problem is to improve the $O(\log^{1.71} n)$ bound for 2-d nearest neighbor search. This might be difficult and require significant new ideas, as we have already attained the seemingly best recurrence one can hope for with the separator approach ($Q(n) \approx 2Q(n^{2/3})$).

Some of the most intriguing open questions for us concern the permutation+bits model. As noted in Section 2.1, even if we allow potentially huge amount of extra space, it is not clear how one can answer nearest neighbor queries in $O(\log n)$ time here. It would be exciting if a superlogarithmic lower bound could be proved in this model, for 2-d nearest neighbor search or some other natural problem. The model is appealing in that it contains features of the bit-probe model but is still "compatible" with traditional computational geometry thinking (that input points form an abstract data type accessible only through comparisons of certain constant-degree algebraic functions).

Many other interesting open problems remain in the area of in-place geometric data structures and algorithms. For example, can one encode an arbitrary planar subdivision (or more specifically a triangulation, or more simply a simple polygon) by a permutation of its vertices so that point location queries can be answered in sublinear time? Is there an in-place algorithm for computing the weight of the Euclidean minimum spanning tree of n points in the plane in subquadratic time?

Is there an in-place algorithm to find the bichromatic closest pair between two planar n-point sets in $o(n \log^2 n)$ time? Can one design a *fully dynamic* in-place data structure for 2-d nearest neighbor search with, say, $O(n^\varepsilon)$ query and update time? (The only known fully dynamic data structure for 2-d nearest neighbor search with polylogarithmic query and update time requires superlinear space [7].)

Finally, while we do not claim that our in-place data structure is directly practical (constant factors might be large due to the repeated usage of planar graph separators), some of the ideas here could still have an impact on implementations. For instance, we can greatly simplify our nearest neighbor method if sublinear extra space is allowed. Also, our data structure is automatically cache-oblivious, with $O(\log_B^{1.71} n)$ query cost for cache size B (although better cache-oblivious, non-in-place results were known).

References

[1] L. C. Aleardi, O. Devillers, and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Proc. 9th Workshop Algorithms Data Struct.*, volume 3608 of *Lecture Notes Comput. Sci.*, pages 134–145, 2005.

[2] L. C. Aleardi, O. Devillers, and G. Schaeffer. Optimal succinct representations of planar maps. In *Proc. 22nd ACM Sympos. Comput. Geom.*, pages 309–318, 2006.

[3] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom. Theory Appl.*, 37:209–227, 2007.

[4] H. Brönnimann and T. M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Comput. Geom. Theory Appl.*, 34:75–82, 2006.

[5] H. Brönnimann, T. M. Chan, and E. Y. Chen. Towards in-place geometric algorithms and data structures. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, pages 239–246, 2004; full version submitted to *ACM Trans. Algorithms*, http://www.cs.uwaterloo.ca/~tmchan/pub.html#inplace.

[6] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoret. Comput. Sci.*, 321:25–40, 2004.

[7] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proc. 17th ACM-SIAM Sympos. on Discrete Algorithms*, pages 1196–1202, 2006.

[8] T. M. Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. Manuscript, 2006, http://www.cs.uwaterloo.ca/~tmchan/pub.html#sss.

[9] T. M. Chan and M. Pătraşcu. Point location in sublogarithmic time and other transdichotomous results in computational geometry. *SIAM J. Comput.*, submitted; preliminary versions in FOCS 2006.

[10] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[12] E. D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes Comput. Sci. to appear, http://erikdemaine.org/papers/BRICS2002/.

[13] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[14] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[15] G. Franceschini and V. Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. In *44th IEEE Sympos. Found. of Comput. Sci.*, pages 242–250, 2003.

[16] G. Franceschini and R. Grossi. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$ time. In *Proc. 14th ACM-SIAM Sympos. on Discrete Algorithms*, pages 670–678, 2003.

[17] G. Franceschini and R. Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th Workshop on Algorithms Data Struct.*, volume 2748 of *Lect. Notes Comput. Sci.*, pages 114–126, 2003.

[18] G. Franceschini, R. Grossi, J. I. Munro, and L. Pagli. Implicit B-trees: New results for the dictionary problem. In *Proc. 43rd IEEE Sympos. Found. of Comput. Sci.*, pages 145–154, 2002.

[19] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. Int. 34th Colloq. Automata, Languages, and Programming*, volume 4596 of *Lecture Notes Comput. Sci.*, pages 533–545, 2007.

[20] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu. Radix sorting with no extra space. In *Proc. 15th European Sympos. Algorithms*, volume 4698 of *Lecture Notes Comput. Sci.*, pages 194–205, 2007.

[21] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31:538–544, 1984.

[22] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41:69–85, 2001.

[23] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[24] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[25] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Sys. Sci.*, 33:66–74, 1986.

[26] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[27] H. Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[28] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[29] R. Seidel and U. Adamy. On the exact worst case query complexity of planar point location. *J. Algorithms*, 37:189–217, 2000.

[30] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[31] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.

[32] J. Vahrenhold. Line-segment intersection made in-place. In *Proc. 9th Workshop Algorithms Data Struct.*, volume 3608 of *Lecture Notes Comput. Sci.*, pages 146–157, 2005.

[33] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6:80–82, 1977.