Towards In-Place Geometric Algorithms and Data Structures*

Hervé Brönnimann† Timothy M. Chan‡ Eric Y. Chen§

February 7, 2006

Abstract

For many geometric problems, there are efficient algorithms that use surprisingly very little extra space other than the given array holding the input. For many geometric query problems, there are efficient data structures that need no extra space at all other than an array holding a permutation of the input. In this paper, we obtain the first such space-economical solutions for a number of fundamental problems, including three-dimensional convex hull and two-dimensional Delaunay triangulation computations, as well as fixed-dimensional range queries and fixed-dimensional nearest neighbor queries.

1 Introduction

As is well known, we can sort an array of n numbers in $O(n \log n)$ time using only a constant amount of extra space, for example, by heapsort. We can also locate a number in a sorted array of n numbers (with no additional structures) in $O(\log n)$ time by binary search. In this paper, we explore analogous *in-place* (or *space-efficient*) algorithms and data structures for problems in computational geometry.

The motivation for saving space is self-evident: in-place (or nearly in-place) algorithms can solve larger problem instances in main memory; in addition, they can be used to handle larger base cases within external-memory divide-and-conquer algorithms.

Although in-place sorting and searching algorithms were investigated a long time ago, there has been a resurgence of interest recently; examples include the latest in-place sorting algorithm [25] that minimizes data movement, and a series of work [26, 27, 28] on in-place (or *implicit*) search structures that are dynamic (allowing insertions and deletions).

In computational geometry, there have been a few recent papers addressing in-place/space-efficient algorithms: Brönnimann *et al.* [8] considered the 2-d convex hull problem, Brönnimann and Chan [6] considered the convex hull of a 2-d polygonal chain, and Chen and Chan (and, more recently, Vahrenhold [46]) considered the segment intersection problem [18] as well as Klee's measure problem in 2-d [19]. Also, Bose *et al.* [5] gave a general technique for implementing stackless

*A preliminary version of this paper appeared in *Proc. 20th ACM Sympos. Comput. Geom.*, pages 239–246, 2004 [7]. Parts of this paper also appeared in the third author's Master's thesis (2004).

†Computer and Information Science, Polytechnic University, Six Metrotech Center, Brooklyn, NY 11201, USA, hbr@poly.edu. Research of this author has been supported by NSF CAREER Grant CCR-0133599.

‡School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, tmchan@uwaterloo.ca. Research of this author has been supported by an NSERC Research Grant and a Premier's Research Excellence Award.

§School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, y28chen@uwaterloo.ca.

divide-and-conquer algorithms, and applied it to closest pair and orthogonal segment intersection problems. However, many basic geometric problems, such as 2-d Voronoi diagrams, have remained unsolved.

In this paper, we present a more comprehensive study of space-efficient geometric algorithms and data structures. (All space bounds below are measured in words, not bits.)

- We give an in-place algorithm for one of the most fundamental geometric problems—computing the convex hull of an array of n points in 3-d. Our algorithm runs in $O(n \log^3 n)$ time with $O(1)$ extra space (see Section 3). A clarification on the model is needed (since the output polyhedron does not naturally fit in an array of size n): our algorithm permutes the array so that the convex hull vertices (the extreme points) occupy a prefix of the array. If the hull edges and facets are desired, they can be printed to a write-only output stream as well.

- The above result automatically implies space-efficient algorithms for a number of geometric problems including 2-d Voronoi diagrams and Delaunay triangulations within the same time and space bounds. For certain applications of Voronoi diagrams, such as finding the bichromatic closest pair or the Hausdorff distance between two n-point sets in 2-d, we can obtain a slightly faster algorithm that runs in $O(n \log^2 n)$ time using $O(\log^2 n)$ extra space (see Section 4).

- We demonstrate surprisingly the possibility of permuting an array of n points in the plane, so that nearest neighbor queries can be answered in $O(\log^2 n)$ time and constant space (see Section 5). This can be seen as a nontrivial generalization of the binary search result in 1-d. (Apparently, permuting into sorted x- or y-order no longer works.)

- More generally, there are in-place data structures with sublinear query time for orthogonal and non-orthogonal range searching queries, ray shooting queries inside convex polytopes, and linear programming queries in any fixed dimension (see Section 5). As an application of these data structures, we can compute the extreme points of an array of n points in $\mathbb{R}^d$ in $O(n^{2-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$ expected time using $O(1)$ extra space. As another application, we also show how to make our space-efficient 3-d convex hull algorithm output-sensitive, with $O(n \log^3 h)$ expected running time and $O(1)$ extra space, where h is the number of extreme points.

Some of our results are derived by modifying known geometric algorithms or combining them with known space-saving tricks; our simplex range searching data structure (being derived from known partition trees) is perhaps one such example. Other results are not so straightforward. For example, our in-place 3-d convex hull algorithm is not based on any of the previous optimal 3-d convex hull algorithms; it involves both new geometric ideas (inspired by known techniques from kinetic computational geometry) as well as array-layout issues (in a way, we are generalizing the in-place heapsort algorithm in 1-d). Another nontrivial example is our 2-d nearest neighbor data structure, which employs planar graph separators in an interesting way.

Furthermore, identifying which algorithms could be made space-efficient can sometimes be a delicate matter. For instance, although Chen and Chan [18] have noticed that Bentley and Ottmann's sweep algorithm for segment intersection can be implemented with little extra space via known so-called "implicit" data structures, a space-efficient version of Fortune's sweep algorithm for 2-d Voronoi diagrams [21, 24] seems more difficult (because of the complexity of the "beach-line" structure).

We end this section with a quick comparison of in-place algorithms with related topics. First, *data stream* algorithms [40] have a similar goal of minimizing storage, but the input comes in a stream and there is no array; although such algorithms can handle much larger data sets, the problems that can be solved exactly in this model are far more limited, even if multiple passes are allowed (space-efficient multi-pass geometric algorithms have recently been explored by Chan and Chen [14]). Second, *sublinear* algorithms [17] also work with a "structure-less" data structure, but here preprocessing is not even allowed; again, this model is more limited but desirable for massive data sets. A closely-related topic is work on compressing and transmitting geometric structures (e.g., Delaunay triangulations [45] and arbitrary triangulations [22] of a 2-d point set) through a permutation of the data; here, a permutation-based structure with few extra bits is similarly desired, although the encoding and decoding algorithms do not have to be in-place.

With space-efficiency carried to an extreme, the closest topic to our approach is *succinct* data structures which require that the asymptotic amount of space they use match the entropy of the class of structures represented. While many succinct data structures were given for arrays, bit-vectors and dictionaries data structures (and more recently for their dynamic versions), very recently Castelli Alleardi *et al.* are the first to provide succinct geometric data structures. In [9], they give succinct representations for triangulations of point sets in the plane, and in [10] also provide dynamic updates.

2 Preliminaries on Space-Saving Tricks

Most data structures need many ($\Omega(n)$) pointers, and any algorithm that uses such structures is automatically not in-place. One of the main tricks in designing space-efficient algorithms is *to store pointers implicitly* by a permutation of a block of data. More precisely, given a block of $s = c \log n$ points $p_1, \ldots, p_s$, we can permute selected pairs of points to encode $s/2$ bits (enough for $\lfloor c/2 \rfloor$ pointers), for example, by the following convention: the i-th bit is interpreted as a 1 if p_{2i-1} is lexicographically smaller than p_{2i}. (We can assume that no two points are identical by an initial sorting step to remove duplicates.)

With this trick, we can for instance implement a doubly-linked list inside an array with only $O(\log n)$ extra space: just divide the given list into blocks of size s and encode $c = 2$ pointers to the successor and predecessor blocks within each block. For example, Brönnimann and Chan [6] used this *implicit linked list* structure in an in-place implementation of Melkman's 2-d convex hull algorithm.

With more effort, Munro [39] showed how to implement a dynamic binary search tree inside an array with only $O(\log^2 n)$ extra space. Updates and searches on n numbers can be performed in $O(\log^2 n)$ time. This *implicit search tree* structure will be used in one of our algorithms in Section 4. (We will ignore the more complicated, recent developments [26, 27, 28].)

Chen and Chan [18] recently observed that the search tree can be *combined with a heap* without increasing the space and time bounds in Munro's method (the total orders associated with the tree and the heap can be different). A tree/heap combination commonly occurs in sweep-based algorithms in computational geometry; the particular application studied by Chen and Chan concerns a space-efficient implementation of Bentley and Ottmann's segment intersection algorithm.

3 3-d Convex Hulls

In this section we present one of the main results of this paper: an efficient in-place algorithm for 3-d convex hulls. The 3-d problem is far more intricate than the 2-d problem (previously considered by Brönnimann *et al.* [8]). None of the textbook algorithms for 3-d convex hulls—for example, divide-and-conquer [42], randomized incremental construction [20], or gift-wrapping [15]—are space-efficient, because they need to work with intermediate hulls. While a 2-d polygon can be easily represented by the order of its vertices, the standard representation schemes for 3-d polyhedra (such as the DCEL and quad-edge structures [21]) all require a large number of pointers.

In the preliminary version of this paper [7], we described a highly nontrivial adaptation of (Chan's "kinetic" reinterpretation [13] of) Preparata and Hong's divide-and-conquer algorithm [42], to show that 3-d convex hulls can be computed in $O(n \log^3 n)$ time using $O(\log^2 n)$ extra space. The key idea there was to represent an intermediate hull simply by a permutation of its vertices (specifically, the so-called shelling order). It turns out that this weaker representation is sufficient to support efficient merging of two vertically separated hulls, via a sweep in dual space. This merging subroutine then leads to a space-efficient divide-and-conquer algorithm.

In this version of the paper, we describe a less complicated algorithm that also happens to use less space, namely $O(1)$. The overall ideas are similar, but instead of using multiple sweeps to merge sub-hulls, we use a single sweep to compute the entire hull directly without recursion (by maintaining a tree structure as we sweep). As a result, the issue of representations of intermediate 3-d sub-hulls becomes moot. To make our algorithm space-efficient, we adopt tricks similar to Chen and Chan's sweep algorithm for segment intersection [18] (although we do not explicitly need implicit search trees).

We remark that while sweeping is popular in computational geometry (for example, in computing 2-d Voronoi diagrams [24]), it is not common knowledge that the paradigm can be applied to 3-d convex hulls (perhaps because the resulting time bound is not quite $O(n \log n)$). Seidel's algorithm for higher-dimensional convex hulls [44], however, can be interpreted as a sweep in dual space.

In the next subsection, we find it convenient to take a kinetic view of the 3-d convex hull problem. In Section 3.2, we present a self-contained description of our algorithm under the traditional (non-space-efficient) setting. Then in Section 3.3, we show that this algorithm can be made in-place. A slower but more easily implementable variant is outlined in Section 3.4.

3.1 A kinetic approach

We assume that the input points are in general position, by standard perturbation techniques. Without loss of generality, it suffices to compute the upper hull. For example, we can first find the extreme points S of the upper hull, then find the extreme points S' of the xy-projection of S by an in-place 2-d convex hull algorithm [8], arrange the array so that a prefix stores $S - S'$, and finally compute the lower hull of the complement of $S - S'$. Alternatively, we can apply the transformation $(x, y, z) \mapsto (x/z, y/z, -1/z)$, which turns the convex hull into an upper hull, assuming (by translation) that the lowest input point is the origin.

As noted in [13], constructing the upper hull of a set of 3-d points is equivalent to maintaining the upper hull of a *kinetic* set of 2-d points that are moving vertically at fixed velocities: For each 3-d point $p_i = (x_i, y_i, z_i)$, the corresponding 2-d point is defined as $\widehat{p_i}(t) = (x_i, z_i - ty_i)$ at time t.

For any three such points $\widehat{p_i}, \widehat{p_j}, \widehat{p_k}$ in left-to-right order, we say that $\widehat{p_i}\widehat{p_j}\widehat{p_k}$ is *convex* at a specified time value t, if it forms a clockwise turn, or *concave* otherwise. Clearly, each such triple can

change from concave to convex (or convex to concave) at only one time t (namely when t satisfies $[(z_1 - ty_1) - (z_2 - ty_2)]/(x_1 - x_2) = [(z_3 - ty_3) - (z_2 - ty_2)]/(x_3 - x_2)$).

As time progresses, a vertex $\widehat{p_j}$ may appear on or disappear from the 2-d upper hull. For example, $\widehat{p_j}$ may be inserted between $\widehat{p_i}$ and $\widehat{p_k}$, as $\widehat{p_i}\widehat{p_j}\widehat{p_k}$ becomes convex; and $\widehat{p_j}$ in between $\widehat{p_i}$ and $\widehat{p_k}$ may be deleted, as $\widehat{p_i}\widehat{p_j}\widehat{p_k}$ becomes concave. Such an insertion or deletion event corresponds precisely to a facet of the 3-d upper hull (namely, the triangle $p_ip_jp_k$). Note that a vertex may be inserted or deleted at most once (the order of insertion is the shelling order). We will compute the 3-d upper hull by tracking the changes to the kinetic 2-d upper hull, as we "sweep" over all time values from $t = -\infty$ to $t = \infty$. Thus, the 3-d problem is reduced to a 2-d data structure problem, which is more susceptible to a space-efficent solution.

We note that much work has been done on the design of kinetic data structures [3, 30]; in particular, the kinetic 2-d convex hull problem was addressed by Basch *et al.* [3] and Alexandron *et al.* [2]. However, our situation is a special case, where points only move vertically at constant velocities, which allows for a more efficient algorithm (the general case may require $\Omega(n^2)$ events).

We remark that the above kinetic viewpoint is mostly a matter of convenience. In the dual, the 3-d upper hull becomes a 3-d lower envelope of planes, and the corresponding 2-d upper hull at a particular time value becomes a 2-d cross-section of this 3-d envelope (a lower envelope of lines that moves linearly over time). Thus, sweeping over time can be reinterpreted as sweeping over one axis in dual space.

3.2 The algorithm

We first describe our algorithm without consideration of space usage. The algorithm is expressed in terms of the kinetic 2-d upper hull problem. (All points in this subsection are 2-d vertically moving points.)

The kinetic data structure we use is a complete binary tree (similar to a "hull tree" data structure of Overmars and van Leeuwen [41]). We first sort all points from left to right (say, by heapsort) and group them into blocks of size s. These blocks are leaf nodes in the binary tree. For each node v, we refer to the upper hull of all points in the leaf blocks underneath v as the *sub-hull* at v. We specify what is stored in each node:

- At each leaf node v, we maintain two lists of points, both in left-to-right order: H_v contains the current sub-hull at v, and H'_v contains the remaining points in the block. Consider the next change in the sub-hull at v, the insertion or deletion of a point; let p_v be this point. We store p_v^-, p_v, and p_v^+, where p^- denotes the rightmost vertex to the left of p, and p^+ denotes the leftmost vertex to the right of p on the sub-hull at v.

- At each internal node v, let l_vr_v denote the *bridge* edge between the sub-hulls at the left child and the right child of v, with l_v on the left sub-hull and r_v on the right sub-hull. We store l_v, r_v, l_v^-, l_v^+, r_v^-, and r_v^+. (See Figure 1.)

We can initialize the data structure at $t = -\infty$ in $O(n \log n)$ time in a bottom-up manner: at the leaves, we can compute the sub-hulls by any optimal 2-d convex hull algorithm; at each internal node, we can compute the bridge by a linear-time scan or a logarithmic-time binary search [41].

To maintain this data structure over time, we consider the following types of events:

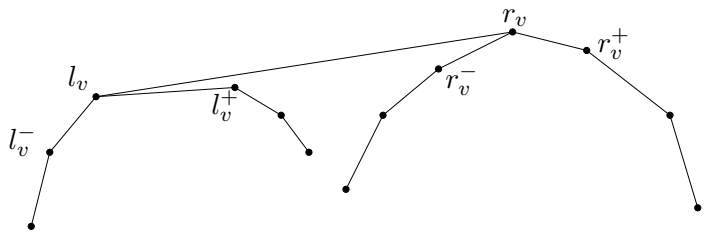

Figure 1: Maintaining the bridge between two sub-hulls.

- Event 1. Time: When $p_v^- p_v p_v^+$ becomes concave for some leaf node v.
  Action (p_v is deleted from the sub-hull at v): Move p_v from H_v to H'_v. Find the next p_v^-, p_v, and p_v^+. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

- Event 2. Time: When $p_v^- p_v p_v^+$ becomes convex for some leaf node v.
  Action (p_v is inserted to the sub-hull at v): Move p_v from H'_v to H_v. Find the next p_v^-, p_v, and p_v^+. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

- Event 3. Time: When $l_v^- l_v r_v$ becomes concave for some internal node v.
  Action (l_v is deleted from the sub-hull at v): $l_v \leftarrow l_v^-$. Find the new l_v^-. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

- Event 4. Time: When $l_v r_v r_v^+$ becomes concave for some internal node v.
  Action (r_v is deleted from the sub-hull at v): $r_v \leftarrow r_v^+$. Find the new r_v^+. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

- Event 5. Time: When $l_v l_v^+ r_v$ becomes convex for some internal node v.
  Action (l_v^+ is inserted to the sub-hull at v): $l_v \leftarrow l_v^+$. Find the new l_v^+. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

- Event 6. Time: When $l_v r_v^- r_v$ becomes convex for some some internal node v.
  Action (r_v^- is inserted to the sub-hull at v): $r_v \leftarrow r_v^-$. Find the new r_v^-. Update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of v.

A priority queue holds the time values of the above events at all leaf/internal nodes v. Our algorithm repeatedly finds the event with the earliest time value, advances the current time to this value, and executes the corresponding action, until $t = \infty$.

We further elaborate on how the actions are carried out.

- In Events 1 and 2, we need to identify the next change to H_v in order to set p_v^-, p_v, and p_v^+ at a leaf node v. This can be done in $O(s)$ time by scanning through H_v and H'_v: For each consecutive triple $p^- p p^+$ in H_v, note the time when $p^- p p^+$ becomes concave. For each consecutive pair pp^+ in H_v and each q in H'_v with q to the right of p and left of p^+, note the time when pqp^+ becomes convex. The earliest time noted is the time of the next change.

- In Events 3–6, we need to recompute p^- (or p^+) of a vertex p of the sub-hull at a child w of an internal node v. This can be done in $O(\log n)$ time by searching downward in the tree: If $p = r_w$, then $p^- = l_w$. If $p = l_w$ or p is to the left of l_w, compute p^- at the left child of w

recursively; otherwise, compute p^- at the right child of w recursively. If w is a leaf node, we can compute p^- directly by a binary search in H_w.

- In all events, we need to update $l_u^-, l_u^+, r_u^-, r_u^+$ at all ancestors u of v that are affected by the change in the sub-hull at v. This can be done by searching upward in the tree: let w be the parent of v and update $l_w^-, l_w^+, r_w^-, r_w^+$; if the change at v occurs to a vertex to the left of l_w or right of r_w, then the change occurs at w as well, so update $l_u^-, l_u^+, r_u^-, r_u^+$ at ancestors u of w recursively; otherwise, stop. If the change occurs at the root, we can print out the corresponding facet; if the change at the root is the insertion of a vertex, we mark it as a output vertex.

At the end, we swap all marked points to a prefix of the array, which takes $O(n)$ time.

For the analysis, observe that there are $O(n)$ changes to the sub-hulls at the nodes of each level of the tree. The total number of events is therefore $O(n \log n)$. Each of the $O(n \log n)$ priority queue updates costs $O(\log n)$ time. Each of the $O(n \log n)$ events at internal nodes requires $O(\log n)$ time to process. Each of the $O(n)$ events at leaf nodes requires $O(s)$ time. The total time of the algorithm is thus $O(ns + n \log^2 n)$.

3.3 An in-place implementation

We now show that the algorithm in Section 3.2 can be made space-efficient, using the encoding ideas from Section 2. We observe that there are $\frac{n}{s}$ leaf nodes and $\frac{n}{s} - 1$ internal nodes in total in the tree (assuming that $\frac{n}{s}$ is a power of 2). We number the leaf nodes $1, \ldots, \frac{n}{s}$ from left to right, and number the internal nodes $1, \ldots, \frac{n}{s} - 1$ from top to bottom and from left to right across each level. We associate each leaf node v with the internal node u of the same number (if v is not the last leaf). At the leaf node v, we encode the points p_v^-, p_v, p_v^+ and $l_u^-, l_u, l_u^+, r_u^-, r_u, r_u^+$. The permutation trick from Section 2 can be used if we set $s = c \log n$ for a sufficiently large constant c.

For the priority queue, we suggest a *tournament tree* implementation [32] instead of a heap. At v, we encode an additional pointer to the node w that defines the earliest event among all events at nodes underneath the associated internal node u. The overall earliest event is thus stored at the root. Each priority queue update can still be carried out using $O(\log n)$ pointer operations.

A remaining problem is how to avoid the extra bit to mark an output vertex. To this end, we partition each leaf block v into four (instead of two) lists, each in left-to-right order, containing: (i) all marked points in H_v, (ii) all unmarked points in H_v, (iii) all marked points in H'_v, and (iv) all unmarked points in H'_v. We encode the sizes of these lists in v as well.

In applying the encoding trick, we have to be careful not to permute pairs across boundaries of the four lists. One solution is to keep all list sizes even by moving the last element of a list to the end of the block if the size of the list is odd. (Because we have encoded the sizes of the original lists, we can easily tell whether the last element of a list has been moved to the end.)

Identifying the next change to the sub-hull at a leaf node v still takes $O(s)$ time, using $O(1)$ space, by scanning through the four lists. Moving a point from H_v to H'_v (or vice versa) or changing a point from unmarked to marked can be done in $O(s)$ time by shifting elements within the block.

Note that we can traverse up/down the tree by simple address arithmetic (like in a heap). We can refer to a point by its index (rank) in the sorted left-to-right order. Given the index, we can locate the actual array entry containing the point in $O(s)$ time, by scanning through the four lists in the relevant block. Encoding/decoding each point or pointer takes $O(\log n)$ time instead of $O(1)$.

The running time in the in-place version is now multiplied by a factor of $O(s+\log n)$. For $s = c\log n$, the overall time bound is therefore $O((ns + n\log^2 n)\log n) = O(n\log^3 n)$. We conclude:

Theorem 3.1 *Given an array of n points in $\mathbb{R}^3$, we can compute the vertices of the upper hull, stored in a prefix of the same array, in $O(n\log^3 n)$ time using $O(1)$ extra space. The edges and facets can also be printed within the same time bound.*

3.4 A more practical space/time trade-off

The implicit encoding/decoding trick makes the algorithm hard to implement. In this subsection, we briefly mention a simpler variant of the algorithm that is slower but still uses sublinear space. (Though suboptimal, it is better than the "brute-force" in-place algorithm that takes cubic time.)

For instance, we can set the block size to $s = \Theta(\sqrt{n})$ and store the tournament tree and the priority queue explicitly (outside the array) in arrays of size $O(n/s)$. Because the tree and the priority queue have size $O(\sqrt{n})$, the extra space needed is $O(\sqrt{n})$. In each leaf block, we can store the sizes of the four lists explicitly. The running time is $O(ns + n\log^2 n) = O(n^{3/2})$.

In fact, this approach can be extended to any block size s such that $\Omega(\log^n) = s \le n$, yielding a wide space/time trade-off:

Theorem 3.2 *Given an array of n points in $\mathbb{R}^3$ and a parameter s such that $c\log^2 n \le s \le n$, for some constant $c > 0$, we can compute the vertices of the upper hull, stored in a prefix of the same array, in $O(ns)$ time using $O(n/s)$ extra space. The edges and facets can also be printed within the same time bound.*

From a practical point of view, the larger the space available, the faster the runtime. Thus a practical variant of the algorithm (so-called *adaptive*) would try to acquire as much space as possible, then set the block size as in Theorem 3.2, and if the available space is less than the required cn/s words, it would then switch to Theorem 3.1.

4 2-d Proximity Problems

Theorem 3.1 implies an in-place $O(n\log^3 n)$ algorithm for 2-d Voronoi diagrams by the standard lifting transformation [38, 43] and can consequently be used to solve a number of off-line proximity problems in 2-d. In this section, we give more direct solutions to these problems that eliminate some of the extra logarithmic factors.

4.1 Monochromatic closest pairs

A number of known algorithms have been proposed for the classic problem of computing the closest pair in a set P of n points in $\mathbb{R}^2$. Some of these algorithms can be made in-place. For example, the well-known divide-and-conquer algorithm in 2-d [43] (the version without "pre-sorting") can be easily implemented in $O(n\log^2 n)$ time and $O(1)$ space (and the approach can be extended to higher fixed dimensions). Using known techniques for in-place merging or stable partitioning, it is possible to reduce the running time of the 2-d algorithm to $O(n\log n)$; this was observed independently by Bose *et al.* [5]. If coordinates are integers in a finite universe $U = [1, \ldots, 2^w]$ and the bitwise exclusive-or operation is supported on words of width w, then there is a simpler solution (which works in any fixed dimension on the word RAM): the "shift-shuffle-and-sort" algorithm suggested by Chan [12] is already in-place (since the problem is directly reduced to sorting).

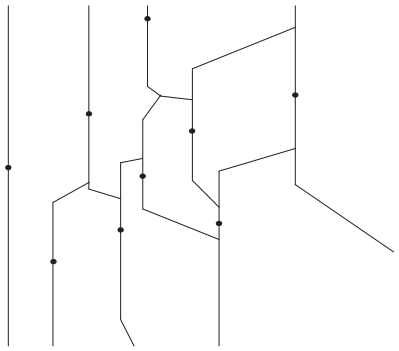

Figure 2: A left Voronoi diagram.

4.2 Bichromatic closest pairs

The bichromatic version of the closest pair problem is perhaps more challenging: given a set P of red points and a set Q of blue points in $\mathbb{R}^2$, we want $\min_{p \in P, q \in Q} \|p - q\|$. Here we suggest a sweep approach, similar to an algorithm by Hinrichs *et al.* [31], described below.

We solve a more general problem: for each blue point q, find the nearest neighbor among all red points to the left of q (call this the *left nearest neighbor* of q). A symmetric subproblem for the right nearest neighbors can be solved similarly, and we can take the minimum of the distances found.

Define the *left Voronoi diagram* of the red points to be the planar subdivision where two points lie in the same cell iff they have the same left nearest neighbor. Note that the cells are not necessarily convex but are x-monotone polygons. (See Figure 2.) The left Voronoi diagram is more susceptible to the sweep paradigm than the ordinary Voronoi diagram, because we know in advance the insertion time of each point: the minimum x-coordinate in the cell of a red point p is precisely the x-coordinate of p.

Here are the details of the sweep for the left Voronoi diagram. We first sort all the red points and all the blue points according to their x-coordinates by heapsort. Let A be the list of all red points to the right of the current vertical sweep line; this list is arranged in increasing order by x-coordinates. Let T of the list of all red points whose left Voronoi cells intersect the sweep line; this list is ordered by y-coordinates. Let p^- and p^+ denote the predecessor and successor of p in the list T.

We initialize $A \leftarrow P$ and $T \leftarrow \emptyset$ at $x = -\infty$. To maintain the lists as the sweep line moves from left to right, we consider the following types of events:

- Event 1. Time: the x-coordinate of the head p of A.
  Action: Remove p from A. Insert p to T. A consecutive group of points may have to be deleted from T as follows: while the circumcenter of pp^+p^{++} is to the left of the sweep line, delete p^+ from T; while the circumcenter of $p^{--}p^-p$ is to the left of the sweep line, delete p^- from T.

- Event 2. Time: the x-coordinate of the circumcenter of p^-pp^+ for some $p \in T$, if the value is to the right of the sweep line.
  Action: Delete p from T.

A priority queue holds the time values of Event 2 for all points in T. Our algorithm repeatedly finds the event with the minimum time value, advances the sweep line to this value, and executes the corresponding action, until $x = \infty$. Whenever the sweep line passes through the x-coordinate of

a blue point q, we can report the left nearest neighbor of q by performing a binary search over T to find the left Voronoi cell containing q.

To obtain a space-efficient implementation, we maintain T using Munro's implicit search tree [39] combined with a heap, as in Chen and Chan's approach [18] mentioned in Section 2. Note that T and A are disjoint at all times. We store the data structure for T in a prefix of the array, with $O(\log^2 n)$ extra space, and store A at the back of the array. There are $O(n)$ updates to T and the priority queue. Each update/search requires $O(\log^2 n)$ time [18, 39]. The total running time of the whole algorithm is thus $O(n \log^2 n)$.

Theorem 4.1 *Given a set P of n red points and a set Q of n blue points in $\mathbb{R}^2$ stored in two arrays, we can compute the bichromatic closest pair in $O(n \log^2 n)$ time with $O(\log^2 n)$ extra space.*

Independently, Bose *et al.* [5] have proposed a *randomized* $O(n \log n)$ algorithm for 2-d bichromatic closest pair using $O(\log n)$ extra space; however, their approach does not yield $o(n)$-space algorithms for the more general problem considered in the next subsection.

4.3 Bichromatic all-nearest-neighbors

A more difficult example is to compute the Hausdorff distance $\max_{q \in Q} \min_{p \in P} \|p - q\|$. Here, it suffices to report the nearest neighbor of every blue point q. We can adopt the same approach as in Section 4.2, but we face one difficulty: one sweep can compute the left nearest neighbors, and a second sweep can compute the right nearest neighbors, but we don't have space to store the result from the previous sweep to determine the overall nearest neighbors. We suggest a more clever algorithm for the second sweep that constructs the right Voronoi diagram and "reconstructs" the left Voronoi diagram *simultaneously*.

The first left-to-right sweep proceeds exactly as in Section 4.2, except that in Event 2, when p is deleted from T, we also insert p to the tail of an output list B. We can store this list B in a prefix of the array, store T in the middle of the array, and A at the back. (At all times, A, T, and B are disjoint.) At the end of the algorithm, we thus obtain a list of all red points B sorted in "order of deletion" (i.e., increasing order with respect to the maximum x-coordinates of the left Voronoi cells). This order will help later when we simulate the sweep of the left Voronoi diagram "backwards" from right to left.

We now explain the second, right-to-left sweep. Let T^L (resp. T^R) be the list of all red points whose left (resp. right) Voronoi cells intersect the sweep line; both lists are ordered by y-coordinates. Let p^{-L} and p^{+L} (resp. p^{-R} and p^{+R}) denote the predecessor and successor in T^L (resp. T^R).

We initialize $T^L \leftarrow P - B$ and $T^R \leftarrow \emptyset$ at $x = \infty$. To maintain the lists as the sweep line moves, we consider the following types of events:

- Event 1. Time: the x-coordinate of some $p \in T^L$.
  Action: Delete p from T^L. Insert p to T^R. A consecutive group of points may have to be inserted to T^L as follows: while the circumcenter of $r^{-L}rr^{+L}$ is to the right of the sweep line for the tail r of B, remove r from B and insert r to T^L.

- Event 2. Time: the x-coordinate of the circumcenter of $p^{-L}pp^{+L}$ for the tail p of B, if the value is to the left of the sweep line.
  Action: Remove p from B. Insert p to T^L.

- Event 3. Time: the x-coordinate of the circumcenter of $p^{-R}pp^{+R}$ for some $p \in T^R$, if the value is to the left of the sweep line.
  Action: Delete p from T^R.

A priority queue holds the time values of Event 1 for all points in T^L; another priority queue holds the time values of Event 3 for all points in T^R. Our algorithm repeatedly finds the event with the maximum time value, advances the sweep line to this value, and executes the corresponding action, until $x = -\infty$. Whenever the sweep line passes through the x-coordinate of a blue point q, we can find the left and right nearest neighbors of q by performing binary searches over both T^L and T^R, and report the closer of the two neighbors.

To obtain a space-efficient implementation, we maintain T^L and T^R in two separate combined implicit search-tree/heap structures [18, 39]. The key observation is that the lists B, T^L, and T^R are disjoint at all times. We can store B in a prefix of the array, and T^L and T^R at the back. There are $O(n)$ updates to T^L, T^R, and the two priority queues. The total running time is again $O(n \log^2 n)$.

Theorem 4.2 *Given a set P of n red points and a set Q of n blue points in $\mathbb{R}^2$ stored in two arrays, we can print the set of pairs $\{(p_q, q)\}_{q \in Q}$, where p_q denotes the nearest red neighbor to q, in $O(n \log^2 n)$ time with $O(\log^2 n)$ extra space.*

As in Section 3.4, a more practical variant with a space/time trade-off can be derived by storing an explicit representation of the search-tree and heap data structures.

5 Range Searching

In this section we demonstrate how certain known geometric data structures can be made in-place and also propose some new in-place data structures. A number of basic geometric problems are considered: halfspace/simplex range searching in fixed dimensions (Sections 5.1–5.3), queries for convex polyhedra (Sections 5.4 and 5.5), and orthogonal range searching (Section 5.6). We also discuss semi-dynamic updates (Section 5.7) and applications to constructing convex hulls (Section 5.8).

5.1 A simple (static) partition tree

We find it instructive to start with an early (suboptimal) method for halfspace range counting in 2-d—Willard's partition tree [47]. Given a static set of n points in the plane, the preprocessing algorithm constructs two lines and recursively handles the subset of points inside each of the four regions formed by the two lines. The first line is taken as the vertical line through the point p_1 with the median x-coordinate. The second line, defined by two points p_2, p_3 on each side of p_1, is chosen such that the number of points in each region formed (excluding p_1, p_2, p_3) is exactly $(n-3)/4$ (ignoring floors and ceilings for simplicity), by the ham-sandwich cut theorem.

The resulting tree structure takes up $O(n)$ space, but a space-efficient version is actually not difficult to obtain. Just store p_1, p_2, p_3 at the head of the array, divide the remaining part of the array into four subarrays of equal size, and recurse in these subarrays. The data structure requires no extra space! To answer a query, we only need to recurse in three of four subarrays; the query algorithm runs in $O(n^{\log_4 3}) = O(n^{0.793})$ time and $O(1)$ space, as simple arithmetic allows us to traverse up/down the tree.

The preprocessing algorithm takes $O(n \log n)$ time, because the ham-sandwich cut problem (the separable case) can be solved in linear time by Megiddo's algorithm [37]. Megiddo's ham-sandwich cut algorithm can actually be made in-place: Brönnimann *et al.* previously observed [8] that Megiddo's 2-d linear programming algorithm can be made in-place, and the same space-efficient prune-and-search strategy works for ham-sandwich cuts as well.

5.2 More implicit partition trees

Naturally we can try the same idea to lay out other kinds of partition trees in an array. Here we consider Matoušek's near-optimal partition trees [34] for simplex range counting in an arbitrary fixed dimension d, which are obtained by the following theorem:

Lemma 5.1 (Partition Theorem) *Given an n-point set P in $\mathbb{R}^d$ and a parameter r, we can partition P into r subsets $\{P_i\}$ each of size $O(n/r)$, and enclose each P_i by a simplex, such that the maximum number of simplices crossed by any hyperplane is bounded by $\kappa = O(r^{1-1/d})$.*

We need a few observations about the construction of the partition, which follow by inspecting Matoušek's proof [34]:

(i) If r is a constant, the randomized version (for example, see [38]) of Matoušek's algorithm is actually in-place and takes $O(n)$ expected time: the algorithm selects a random sample R of about $r^{1/d}$ points, and works with "test" hyperplanes that are defined by d points of R; the amount of extra space depends only on r.

(ii) Each subset P_i except the last has size exactly $\lfloor n/r \rfloor$.

(iii) Each simplex is defined by a constant (depending only on d) number of points in R: specifically, the algorithm uses only simplices with vertices that come from the arrangement of the test hyperplanes.

A space-efficient version of Matoušek's partition tree can be obtained as follows: Pick a subset S of size $O(r^2)$ so that each subset $P_i - (R \cup S)$ except the last has exactly the same size. Store $R \cup S$ at the head of the array, divide the remaining array into r subarrays, and recursively store $P_i - (R \cup S)$ in the i-th subarray. Because of (iii), the r simplices can be encoded by permuting pairs in S (as in Section 2) as long as $|S| = \omega(r \log r)$. The data structure needs no extra space and, like Matoušek's original partition tree, can answer simplex range counting queries in $O(n^{1-1/d+\varepsilon})$ time and $O(1)$ space for any fixed $\varepsilon > 0$, by making r a sufficiently large constant.

The preprocessing algorithm is in-place and takes $O(n \log n)$ total expected time by (i), since r is a constant.

Theorem 5.2 *Given an array of n points in $\mathbb{R}^d$ and any fixed $\varepsilon > 0$, we can permute the array in $O(n \log n)$ expected time, using $O(1)$ space, so that we can count the number of points in a query simplex with $O(n^{1-1/d+\varepsilon})$ time and $O(1)$ space. We can also report all k points in the simplex in $O(k)$ additional time.*

5.3 Finer partitions

Matoušek [34] gave slight further improvements to his data structure that lower the extra n^ε factor by using partitions with a nonconstant parameter r. We show that improvements are possible in the in-place setting as well; however, the array layout becomes much more intricate, as we shall see.

We first invoke the partition theorem with $r = \lfloor n/\log^a n \rfloor$, where a is a suitable constant. We divide the array into blocks of size $\Theta(\log^a n)$ holding the subsets P_i's.

Matoušek [34, in the proof of Theorem 6.1] gave an auxiliary data structure to store the r simplices, with $O(r \operatorname{polylog} r)$ preprocessing time, so that we can (i) count the number of simplices contained in a query simplex q, and (ii) report all simplices crossing the boundary of q, both in $O(r^{1-1/d} \operatorname{polylog} r + \kappa)$ time, where κ is the crossing number bound (an upper bound on the number of answers to query (ii)). The space required by this data structure is $O(r \operatorname{polylog} n)$ words or bits. Since this number is less than $n/2$ for a sufficiently large a, our key idea is to encode the entire auxiliary data structure inside the input array by permuting pairs of points within all the blocks. Time bounds are now increased by a factor of $O(\log n)$, because of the need to encode/decode words in each step. The simplices themselves can be encoded in the array as well, since these require only $O(r \log n)$ extra bits (as noted in Section 5.2, each simplex can be specified by only a constant number of input points).

To count the number of points in a query simplex q, we first count the number of simplices (excluding the last one) that are contained in q and multiply the number by $\lfloor n/r \rfloor$. We then identify all simplices crossing the boundary of q, and for each such simplex (plus the last one), examine the points inside the corresponding block and increment the count for each point that lies in q. The number of points examined is $O(r^{1-1/d} \operatorname{polylog} n)$ times $\Theta(\log^a n)$, with $r = \lfloor n/\log^a n \rfloor$. The overall query time is thus $O(n^{1-1/d} \operatorname{polylog} n)$.

The preprocessing requires the construction of a partition for a large value of r. Matoušek [34, Theorem 4.7(ii)] showed that this can be done in $O(n \log n)$ time (via a recursive application of the partition construction for small r), if the crossing number bound κ is increased by a polylogarithmic factor. However, this algorithm requires $O(n)$ space. If we insist on an in-place preprocessing algorithm, we can do the following: Arbitrarily divide P into b subsets $\{S_j\}$ each of size at most $\lceil n/b \rceil$, where $b = \lceil c \log n \rceil$ for a suitable constant c. Build our in-place data structure for S_j. The space needed for this preprocessing is technically $O((n/b)\log(n/b))$ bits. Since this number is less than $(n - \lceil n/b \rceil)/2$ for a sufficiently large c, however, we can simulate the algorithm without extra space by permuting pairs in $P - S_j$. The running time is increased by a factor of $O(\log n)$, due to the encoding/decoding of words. So, the overall preprocessing time is $O(n \log^2 n)$. We can answer a query for P by answering a query for each S_j; the query time is increased by at most a factor of $b = O(\log n)$.

Theorem 5.3 *With $O(n \operatorname{polylog} n)$ preprocessing time, we can replace the n^ε factor by $\operatorname{polylog} n$ in Theorem 5.2.*

5.4 Polytope queries

A similar approach can be taken for Matoušek's data structure for halfspace emptiness queries [35] (deciding whether a query halfspace contains any input point), which replaces the exponent $1-1/d$ by $1-1/\lfloor d/2 \rfloor$, for $d \geq 4$. This uses a variant of the partition theorem for "shallow hyperplanes". The improvement in Section 5.3 is also applicable, using the appropriate auxiliary data structure (as provided by Matoušek [35, in the proof of Theorem 1.3]). The query time becomes $O(n^{1-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$.

Any data structure for halfspace emptiness queries can be used to answer ray shooting and linear programming queries in the dual polytope by parametric search [1, 36]. So, we automatically obtain results for these polytope queries. There is one slight issue, though: the space required by the query algorithm is now $O(n^{1-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$ because of the application of parametric search. Since this number is $o(n)$, one idea is to reserve, say, $n/10$ pairs in the array that we can permute (which can be made available in the method of Section 5.3 by adjusting the constants), so that we can simulate the query algorithm without needing space outside the array. The query time is increased by another logarithmic factor due to the encoding/decoding of words.

Theorem 5.4 *Given an array of n halfspaces in $\mathbb{R}^d$ with $d \geq 4$, we can permute the array in $O(n \operatorname{polylog} n)$ time, using $O(1)$ space, so that we can answer ray shooting queries and linear programming queries in the intersection of the halfspaces in $O(n^{1-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$ time and $O(1)$ space.*

We can obtain similar results for nearest neighbor queries for a point set in $\mathbb{R}^{d-1}$, as these queries can be reduced to ray shooting in $\mathbb{R}^d$ by the standard lifting transformation [38, 43].

5.5 3-d polytopes

For polytope queries in 3-d, we describe a new strategy that uses planar graph separators instead of Matoušek's partitions. We need an extension of Lipton and Tarjan's planar separator theorem [33], as given by Frederickson [29] and restated below in a dual form.

Lemma 5.5 (Separator Theorem) *Given a planar graph with n faces and a parameter b, we can partition the faces into a* separator *of $O(n/\sqrt{b})$ faces and $O(n/b)$* clusters *of $O(b)$ faces each, such that no two different clusters share an edge and each cluster is adjacent to $O(\sqrt{b})$ separator faces. The construction can be done in $O(n \log n)$ time and space.*

We first compute the boundary $\mathcal{P}_H$ of the intersection of the given set H of n halfspaces in $\mathbb{R}^3$, triangulate all faces, then invoke the separator theorem with $b = \lfloor c \log^2 n \rfloor$, where c is a suitable constant. Let S be the subset of all halfspaces that define separator faces. Let H_i be the subset of all halfspaces not in S that define faces in the i-th cluster, and H_i' be the subset of all halfspaces in S that define faces in the i-th cluster. Note that the H_i's are disjoint, $|H_i| = O(b)$, $|H_i'| = O(\sqrt{b})$, and $|S| = O(n/\sqrt{b})$. We divide the array into $O(n/b)$ blocks, where the i-th block stores H_i and the last block stores S.

Next we compute the boundary $\mathcal{P}_S$ of the intersection of S, triangulate so that the separator faces (triangles) in $\mathcal{P}_H$ also are faces in $\mathcal{P}_S$, then store $\mathcal{P}_S$ under Dobkin and Kirkpatrick's hierarchical data structure [23]. The space required by the data structure is $O(|S|)$ words, i.e., $O((n/\sqrt{b}) \log n)$ bits. Since this number can be made smaller than $n/2$ for a sufficiently large c, our key idea is to encode the entire data structure by permuting pairs in the array. Note that the same separator faces also divide $\mathcal{P}_S$ into the same number of clusters; we make each interior vertex and face inside a cluster of $\mathcal{P}_S$ point to the cluster. We also store the starting positions of the blocks and the subsets H_i'. These extra information requires $O((n/\sqrt{b}) \log n)$ bits and can be encoded in the array as well.

Given a line q, suppose we want to find the leftmost intersection of q with $\mathcal{P}_H$, denoted $\textsc{ans}(H, q)$ (a ray shooting query). We first compute $\textsc{ans}(S, q)$ by querying Dobkin and Kirkpatrick's data structure for $\mathcal{P}_S$. Ordinarily, this takes $O(\log n)$ time [23], but because of encoding/decoding, the bound is increased to $O(\log^2 n)$. If $\textsc{ans}(S, q)$ lies in a separator face, then we return $\textsc{ans}(S, q)$. If

$\text{ANS}(S, q)$ lies in the i-th cluster of $\mathcal{P}_S$, then we return $\text{ANS}(H_i \cup H_i', q)$ by examining the intersections of q with all halfspaces in the block containing H_i in $O(b)$ time, and with all halfspaces in the encoded subset H_i' in $O(\sqrt{b} \log n)$ time. The overall query time is thus $O(\log^2 n)$.

Given a direction q, we can also find the extreme vertex of P along q (a linear programming query) by the same strategy. The only difference is that in computing $\text{ANS}(H_i \cup H_i', q)$, we need to run a linear programming algorithm on $O(b)$ halfspaces. This requires $O(\log^2 n)$ time and space. (Probably, the space bound can be reduced with more effort.)

In apply the encoding trick, we again have to be careful not to permute pairs across boundaries of the blocks (especially as the blocks have different sizes). One solution is to keep all block sizes even by moving one halfspace from H_i to H_i' whenever $|H_i|$ is odd.

Theorem 5.6 *Given an array of n halfspaces in $\mathbb{R}^3$, we can permute the array in $O(n \log n)$ time, so that we can answer ray shooting queries in the intersection of the halfspaces in $O(\log^2 n)$ time and $O(1)$ space, and linear programming queries in $O(\log^2 n)$ time and space. If we require the preprocessing algorithm to be in-place, both the preprocessing and query time bounds are increased by a $\log n$ factor.*

As an application, we obtain a similar result for nearest neighbor queries in $\mathbb{R}^2$.

5.6 Implicit kd-trees

For orthogonal range queries over a point set in $\mathbb{R}^d$ (counting/reporting all input points inside a query hyperrectangle), we can use the kd-tree [21], which is even easier to made in-place than Willard's partition tree in $O(n \log n)$ time: store a single point (median along one of the axes) at the head of the array, divide the remaining points into two halves, and recurse in the two subarrays. Reporting k points takes time $O(n^{1-1/d} + k)$ [21]. A partial match query (when $s < d$ coordinates are specified in the query range) takes time $O(n^{1-s/d} + k)$.

Note that by Chazelle's lower bound [16], polylogarithmic query time cannot be attained by any data structure that uses space $o(n(\log n/\log\log n)^{d-1})$, in particular, any in-place data structure. (For example, no in-place version of range trees [21] is possible.)

However, for three-sided orthogonal range searching in 2-d (reporting all points inside a rectangle of the form $(-\infty, q_x] \times [q_y, q_y']$), we can obtain an in-place data structure with query time $O(\log n + k)$ by a (slightly modified) priority search tree [21]: store the point p_1 with the smallest x-coordinate and the point p_2 with the median y-coordinate at the head of the array, divide all other points (excluding p_1 and p_2) into two halves by the median y-coordinate, and recurse in the two subarrays.

5.7 Semi-dynamization

All of the above data structures can also be made to support insertions, by using a standard technique due to Bentley and Saxe [4]. The idea is to partition the data set into $O(\log n)$ subsets, each of size equal to a power of 2, where a subset of size 2^i exists iff the $(i+1)$-st least significant bit in the binary representation of n is 1. To perform an insertion, we increment the number n in binary, identify which bits are turned to 0, and merge the corresponding subsets by building a static data structure for their union. We observe that this process can be easily done in-place if the static data structures are in-place: just put the subsets in decreasing order of size in the array.

For decomposable queries like range searching, the query time is increased by at most a logarithmic factor through this process; if the preprocessing time for the static data structure is $O(n \log n)$,

as in all of our algorithms, the amortized update time is $O(\log^2 n)$. Alternatively, by using a base larger than 2, we can reduce to amortized update time to $O(\log n)$ at the expense of increasing the query time by an n^ε factor. For non-decomposable queries like linear programming, we apply the parametric-search reduction to halfspace emptiness first before applying the above technique.

Supporting deletions appears much harder, if we want to keep the number of cells used by the data structure equal to the current number of elements at all times. We leave this as an open problem (this seems to call for some geometric generalization of dynamic implicit search trees [39]). Likewise, supporting worst-case $O(\operatorname{polylog} n)$ update time in-place, even for insertions only, seems much harder, and we leave this as an open problem.

5.8 Applications to convex hulls

We describe one application of the preceding data structures to finding extreme points in higher dimensions.

Corollary 5.7 *Given an array of n points in $\mathbb{R}^d$ with $d \geq 4$, we can compute the vertices of the convex hull, stored in a prefix of the same array, in $O(n^{2-1/\lfloor d/2 \rfloor} \operatorname{polylog} n)$ expected time using $O(1)$ extra space.*

Proof: Testing whether a point is extreme can be reduced to a linear programming query, so we can apply Theorem 5.4. One problem arises, though: we do not have extra space to "mark" points in the array as extreme. To get around the problem, we borrow an idea from Section 3.3. Recall that the data structure from Theorem 5.4 (based on Section 5.3) divides the array into blocks. We subdivide each block into two lists, one containing all currently marked points in the block, the other containing all unmarked points. We encode in addition the size of these lists in the block. (Like before, to ensure that we do not permute pairs across the boundary of the two lists, we can move the last element of a list to the end of the block if the size of the list is odd.) At the end, we can swap all marked points to a prefix of the array by one linear scan. □

For another application of the in-place data structures, we derive an output-sensitive version of Theorem 3.1 for 3-d convex hulls, by modifying Chan's "group-and-wrap" algorithm [11]. Previously, Brönnimann *et al.* [8] considered space-efficient output-sensitive results for 2-d convex hulls.

Corollary 5.8 *Given an array of n points in $\mathbb{R}^3$, we can compute the h vertices of the lower hull, stored in a prefix of the same array, in $O(n \log^3 h)$ time using $O(1)$ extra space. The edges and facets can also be printed within the same time bound.*

Proof: It suffices to find the extreme points, as the edges and facets can be subsequently generated by Theorem 3.1. For simplicity, we assume that h is known; a standard "guessing" trick (e.g., see [11]) can be applied otherwise. If $h \geq n^{1/4}$, then the bounds in Theorem 3.1 are good enough. So assume $h < n^{1/4}$. We divide the input array into one block of size h and $O(n/h^4)$ blocks of size $O(h^4)$, and build an in-place data structure for dual ray shooting queries for each block except the first, by Theorem 5.6. The total preprocessing time is $O((n/h^4) \cdot h^4 \log^2(h^4)) = O(n \log^2 h)$, and no extra space is used.

We then repeatedly enlarge a connected subset S of hull vertices, stored in a prefix of the first block, as follows. Initially, put the endpoints of any hull edge in S (an initial hull edge can be easily

found in linear time without extra space). In each iteration, we go through each pair $p, q \in S$, test whether pq forms a hull edge, and if so, wrap the hull around $\overrightarrow{pq}$ by a dual ray-shooting query, to get a vertex r; if $r \notin S$, we insert r to S by swapping r with a point in the first block and rebuilding the data structure for the block previously containing r. We quit the algorithm when no insertions to S occur in an iteration.

As the number of iterations is bounded by h, the total number of edge tests and wrapping queries is bounded by $O(h^3)$. Each such query can be reduced to ray shooting in the dual polytope and can be answered by answering a ray shooting query for each subset. Since a query for one subset requires $O(\log^3 h)$ time by Theorem 5.2 (except for the first block, which requires $O(h)$ time by brute force), the total cost of all $O(h^3)$ queries is $O(h^3 \cdot [h + (n/h^4)\log^3(h^4)]) = O(n)$. In addition, $O(h)$ rebuilding operations are performed, each requiring $O(h \log^2 h)$ time. The total running time after preprocessing can therefore be bounded by $O(n)$. □

6 Open Problems

We close with a few intriguing questions that arise from our work:

- Is there an in-place $O(n \log n)$-time algorithm for convex hulls in 3-d?
- Is there a data structure that requires no extra space and can answer 2-d nearest neighbor queries in logarithmic time?
- Is there an $\Omega(\sqrt{n})$ lower bound on the complexity of 2-d orthogonal range queries for (static) data structures with no extra space?
- Can one decide whether a point lies in a simple polygon in sublinear time if the polygon's vertices are preprocessed and stored in a suitable permutation?

References

[1] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.

[2] G. Alexandron, H. Kaplan, and M. Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *Proc. 9th Workshop Algorithms Data Struct.*, volume 3608 of *Lecture Notes Comput. Sci.*, pages 269–281, 2005.

[3] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *J. Algorithms*, 31:1–28, 1999.

[4] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *J. Algorithms*, 1:301–358, 1980.

[5] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Comput. Geom. Theory Appl.*, to appear.

[6] H. Brönnimann and T. M. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. In *Proc. Latin American Theoretical Informatics*, volume 2976 of *Lecture Notes Comput. Sci.*, pages 162–171, 2004. *Comput. Geom. Theory Appl.*, to appear. Available online, Jan. 18, 2006.

[7] 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.

[8] 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.

[9] L. Castelli Alleardi, O. Devillers, and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Proc. Workshop Algorithms Data Structures (WADS)*, volume 3608 of *Lecture Notes Comput. Sci.*, pages 134–145, 2005.

[10] L. Castelli Alleardi, O. Devillers, and G. Schaeffer. Dynamic updates of succinct triangulations. In *Proc. 17th Canad. Conf. Comput. Geom.*, pages 135–138, 2005.

[11] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16:361–368, 1996.

[12] T. M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13rd ACM-SIAM Sympos. on Discrete Algorithms*, pages 472–473, 2002.

[13] T. M. Chan. A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm. Manuscript, `http://www.cs.uwaterloo.ca/~tmchan/pub.html#ch3d`, 2003.

[14] T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 180–189, 2005.

[15] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17(1):78–86, January 1970.

[16] B. Chazelle. Lower bounds for orthogonal range searching, I: The reporting case. *J. ACM*, 37:200–212, 1990.

[17] B. Chazelle, D. Liu, and A. Magen. Sublinear geometric algorithms. In *Proc. 35th ACM Sympos. Theory Comput.*, pages 531–540, 2003.

[18] E. Y. Chen and T. M. Chan. A space-efficient algorithm for segment intersection. In *Proc. 15th Canad. Conf. Comput. Geom.*, pages 68–71, 2003.

[19] E. Y. Chen and T. M. Chan. Space-efficient algorithms for Klee's measure problem. In *Proc. 17th Canad. Conf. Comput. Geom.*, pages 38–41, 2005.

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

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

[22] M. Denny and C. Sohler. Encoding a triangulation as a permutation of its point set. In *Proc. 9th Canad. Conf. Comput. Geom.*, pages 39–43, 1997.

[23] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes Comput. Sci.*, pages 400–413. Springer-Verlag, 1990.

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

[25] G. Franceschini and V. Geffert. An in-place sorting with $O(n \log n)$ comparisons and $O(n)$ moves. *J. ACM*, 52(4):515–537, 2005.

[26] 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.

[27] G. Franceschini and R. Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th Workshop on Algorithms Data Struct.*, Lect. Notes Comput. Sci., 2003.

[28] 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.

[29] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[30] L. J. Guibas. Kinetic data structures: A state of the art report. In *Proc. 3rd Workshop on Algorithmic Foundations of Robotics*, 1998.

[31] K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbors problems. *Acta Inform.*, 29(4):383–394, 1992.

[32] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

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

[34] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[35] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2(3):169–186, 1992.

[36] J. Matoušek and O. Schwarzkopf. Linear optimization queries. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 16–25, 1992.

[37] N. Megiddo. Partitioning with two lines in the plane. *J. Algorithms*, 6:430–433, 1985.

[38] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[39] 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.

[40] S. Muthukrishnan. Data streams: Algorithms and applications. Manuscript, `http://www.cs.rutgers.edu/~muthu`, 2003.

[41] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[42] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

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

[44] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th Annu. ACM Sympos. Theory Comput.*, pages 404–413, 1986.

[45] J. Snoeyink and M. van Kreveld. Linear-time reconstruction of Delaunay triangulations with applications. In *Proc. Annu. European Sympos. Algorithms*, number 1284 in Lecture Notes Comput. Sci., pages 459–471. Springer-Verlag, 1997.

[46] 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.

[47] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.