Towards In-Place Geometric Algorithms and Data Structures

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

December 11, 2003

Abstract

For many geometric problems, there are efficient algorithms that surprisingly use 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 hulls, two-dimensional Delaunay triangulations, 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 initiated a long time ago, there has been a resurgence of interest recently; examples include the latest in-place sorting algorithm [14] that minimizes data movement, and a series of work [15, 16, 17] on in-place (or *implicit*) search structures that are dynamic (allowing insertions and deletions).

In computational geometry, there have been three recent papers addressing in-place/space-efficient algorithms: Brönnimann et al. [3] considered the 2-d convex hull problem, Brönnimann and Chan [2] considered the convex hull of a 2-d polygonal chain, and Chen and Chan [10] considered the segment intersection problem. 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.

*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.

- We give a nearly 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 only $O(\log^2 n)$ extra space (see Section 2). 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.

  A slower but simpler variant of the algorithm is also described.

- The above result automatically implies space-efficient algorithms for a number of geometric problems including 2-d Voronoi diagrams and Delaunay triangulations. For certain applications of the 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 3).

- We demonstrate surprisingly the possibility of permuting an array of n points in the plane so that nearest neighbor queries can be answered in $O(n^\varepsilon)$ time and constant space (see Section 4), where $\varepsilon > 0$ denotes an arbitrarily small constant throughout this paper. The preprocessing algorithm is randomized and runs in $O(n \log n)$ expected time and also uses just $O(1)$ space. 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.)

  Our data structure can also support insertions in $O(\log n)$ amortized time.

- More generally, there are in-place data structures with sublinear query time for (orthogonal and non-orthogonal) range searching queries, ray shooting queries, and linear programming queries in any fixed dimension (see Section 4). As an application of these data structures, we can compute the extreme points of an array of n points in $\mathbb{R}^d$ in $O(n^{2-1/\lfloor d/2 \rfloor+\varepsilon})$ expected time and $O(1)$ extra space. As another application, we also show how to make our space-efficient 3-d convex hull algorithm output-sensitive, with expected running time $O(n \log^3 h)$ and space $O(\log^2 h)$, 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 nearest neighbor data structure is perhaps one such example (based on the partition trees of Matoušek [20, 21]).

However, not all the results are straightforward. The most notable is our 3-d convex hull algorithm. It is based on the standard divide-and-conquer algorithm by Preparata and Hong [27], where merging is done using a dual sweep algorithm, as described in [5]. Although Chen and Chan [10] have shown the possibility of implementing sweep-based algorithms with little extra space via known implicit data structures, our case is much more involved and not only uses a complex data structure layout (in a way, we are generalizing an in-place mergesort algorithm) but requires some new and substantial geometric ideas on the convex-hull merging process as well, which are interesting in their own right.

Furthermore, identifying which algorithms could be made space-efficient can sometimes be a delicate matter. For instance, unlike Bentley and Ottmann's well-known sweep algorithm for segment intersection, an in-place version of Fortune's sweep algorithm for 2-d Voronoi diagrams [11, 13] appears 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 [26] have a similar goal of minimizing storage, but 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 (for example, consider the 2-d Voronoi diagram). Second, *sublinear* algorithms [8] 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. Perhaps the closest topic is work on transmitting geometric structures (Delaunay triangulations [30] and arbitrary triangulations [12] of a 2-d point set) through a permutation of the data; here, a structure with few extra bits is similarly desired, although the preprocessing/encoding and decoding algorithms do not have to be in-place but must run fast.

2 3-d Convex Hulls

In this section we give a nontrivial, space-efficient adaptation of Preparata and Hong's divide-and-conquer algorithm for 3-d convex hulls [27, 28]. Our version of the algorithm is closer to Chan's "kinetic" description [5] but still involves substantial modifications. We first discuss preliminaries on space-efficient data structuring techniques in Section 2.1 and one simple convex-hull representation scheme in Section 2.2. We then describe the main convex-hull merging algorithm in Section 2.3 and the end result in Section 2.4. A slower but more easily implementable variant is outlined in Section 2.5.

2.1 Tools from implicit data structures

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 $s = 2c \log n$ points $p_1, \ldots, p_s$, we can permute selected pairs of points to encode $s/2$ bits (enough for c pointers), for example, by the following convention: the i-th bit is interpreted as a 1 if p_{2i-1} is left of p_{2i}.

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 [2] used this *implicit linked list* structure in an in-place implementation of Melkman's 2-d convex hull algorithm.

With more effort, Munro [25] 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 repeatedly in our algorithm. (We will ignore the more complicated, recent developments [15, 16, 17].)

Chen and Chan [10] 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. This modified implicit search tree will also be needed in our algorithm.

2.2 Representing the hull by its shelling order

Most 3-d convex hull algorithms need to work with intermediate hulls, but the standard polyhedral representation schemes (such as the DCEL and quad-edge structures [11]) all require a large number of pointers. In implementing the divide-and-conquer algorithm, Chan [5] recently suggested an alternative representation scheme. To describe this alternative, we view the problem kinetically in 2-d as follows.

Without loss of generality, it suffices to consider the lower hull, as a projective transformation can be applied to move the lowest point to infinity. As noted in [5], constructing the 3-d lower hull is equivalent to tracking the changes to the lower hull $\mathrm{LH}(P)$ of a set P of points that are moving vertically at fixed velocities in the plane. (This is easily seen in the dual by sweeping: constructing a 3-d halfspace intersection is equivalent to tracking the changes to an intersection of linearly moving halfplanes in 2-d.) Chan [5] suggested storing the sequence of insertion and deletion events undergone by $\mathrm{LH}(P)$, where each deletion event is specified by the point being deleted, and each insertion event is specified by the point being inserted and pointers to the location of the insertion (i.e., the two neighbors in the 2-d hull).

Here we adopt an even simpler scheme: ignore the deletion events and location pointers. In other words, just record the insertion order of the vertices (a permutation, which needs no extra space!). In terms of the original 3-d problem, this insertion order corresponds to the so-called *shelling order*. The deletion order of the vertices, on the other hand, corresponds to what we will call the *backward shelling order*.

Although we can no longer decode the polyhedron in linear time from just the insertion order, the decoding can still be performed in $O(n \operatorname{polylog} n)$ time, with little extra space, as the lemma below indicates. The proof is a nice warm-up exercise of the techniques from Section 2.1.

Lemma 2.1 *Given the shelling order for n points in $\mathbb{R}^3$, stored in an array, we can print the edges and facets of the lower hull in $O(n\log^2 n)$ time using only $O(\log^2 n)$ additional space. The array stores the backward shelling order at the end.*

Proof: As explained above, the problem is equivalent to printing the changes to a kinetic 2-d hull $\mathrm{LH}(P)$, given the insertion order as input.

The algorithm requires at most $2n$ iterations. Straightforward initialization and termination details are omitted here. At each iteration, we identify the current point p in the input list and advance the current time to the the smallest value among the following: the time when (1) p^-pp^+ becomes convex, and the time when (2) q^-qq^+ becomes concave for each vertex q of $\mathrm{LH}(P)$. Here, v^+ and v^- denote the current successor and predecessor of v in $\mathrm{LH}(P)$ (in terms of the left-to-right order). If the smallest time is given by (1), we insert p to $\mathrm{LH}(P)$ (between p^- and p^+) and move on to the next point in the input list; if the smallest is given by (2), we delete q from $\mathrm{LH}(P)$ and move q to an output list that will eventually contain the entire deletion order.

At any time, the array is divided into three parts: the suffix holds the remaining input list, the prefix holds the current output list, and the middle holds the current hull $\mathrm{LH}(P)$. We store the vertices of $\mathrm{LH}(P)$ in left-to-right order using Munro's implicit search tree [25]; this enables us to find successors and predecessors quickly. In addition, we overlay the structure with a heap, following Chen and Chan [10], where the priority value of a vertex q is the time when q^-qq^+ becomes concave; this enables us to find the next time value quickly. The space and time bounds of the algorithm now follow from [10, 25]. □

Note that by running the above algorithm twice, we can get back the original shelling order in the array. The shelling order was most notably used for constructing higher-dimensional convex hulls by Seidel [29] (whose algorithm can also be interpreted as a sweep in dual space). In the next subsection, we show how the shelling order (forward or backward) is an adequate representation for a space-efficient implementation of the 3-d divide-and-conquer algorithm.

2.3 The merging algorithm

The main subproblem in the divide-and-conquer approach is how to merge two linearly separated hulls in 3-d. In the kinetic 2-d setting, we have two sets P_L and P_R of points moving vertically at fixed velocities in the plane, with all points in P_L left of all points in P_R. We have determined the insertion orders for $\mathrm{LH}(P_L)$ and for $\mathrm{LH}(P_R)$. We want to compute the deletion order for $\mathrm{LH}(P)$ where $P = P_L \cup P_R$ (as the deletion order is more convenient than the insertion order).

This merging process is relatively straightforward, as noted already by Chan [5]: all we have to do is to keep track of changes to the bridge (a single edge) between the two 2-d hulls $\mathrm{LH}(P_L)$ and $\mathrm{LH}(P_R)$. However, subtle difficulties arise when we attempt to make the algorithm space-efficient. For example, when a vertex is deleted from the overall hull $\mathrm{LH}(P)$ but is still on the subhull $\mathrm{LH}(P_L)$, we need to place the vertex in an output list and are thus forced to delete it from P_L prematurely. These premature deletions may in turn cause premature insertions to $\mathrm{LH}(P_L)$ (due also to the fact that actual insertion times cannot be stored). Because of these difficulties, we can only maintain some "approximation" $\mathrm{LH}(Q_L)$ to the subhull $\mathrm{LH}(P_L)$. The new algorithm is hence more complicated, with a nontrivial proof of correctness, in addition to the need of the data structuring techniques from Section 2.1.

Lemma 2.2 *Given the shelling orders for two linearly separated sets of $n/2$ points in $\mathbb{R}^3$, stored in an array, we can compute the backward shelling order for the union of the two sets, stored in the same array, in $O(n \log^2 n)$ time using $O(\log^2 n)$ extra space.*

Proof: Because of space limitation, the detailed proof is presented in the appendix. □

2.4 The divide-and-conquer algorithm

We can now put together a space-efficient algorithm for 3-d convex hulls, by heapsorting the given points by x-coordinates, applying Lemma 2.2 recursively, and applying Lemma 2.1 to print the output:

Theorem 2.3 *Given an array of n points in $\mathbb{R}^3$, we can compute the vertices of the lower hull, stored in a prefix of the same array, in $O(n \log^3 n)$ time using $O(\log^2 n)$ extra space. The edges and facets can also be printed within the same time bound.*

2.5 A more practical variant?

The usage of implicit search trees makes the algorithm difficult to implement. In this subsection, we briefly mention a variant of the algorithm that is slower (running in $O(n^{3/2})$ time) but still uses sublinear ($O(\sqrt{n})$) space and is much easier to implement. (Though suboptimal, it is better than the "brute-force" in-place algorithm that takes cubic time.) This new algorithm can only print the result to an output stream, however, and cannot store the vertices in a prefix of the array.

Instead of divide-and-conquer, the idea is to make only one sweep but maintain a binary tree of bridges as time progresses (in terminologies from the 2-d kinetic setting). To keep space low, we only store a tree of $O(\sqrt{n})$ bridges, where leaves correspond to blocks of $O(\sqrt{n})$ points. In each leaf block, we store the vertices of the lower hull from left to right in a prefix of the block. Events associated with bridges are kept in a heap, so that each such event can be processed in $O(\log n)$ time. The next event associated with a leaf is generated "from scratch" (by re-scanning the hull edges and points in the block from left to right), costing $O(\sqrt{n})$ time each. Over time, there are $O(n \log n)$ bridge events and $O(n)$ leaf events, so that the total running time is bounded by $O(n^{3/2})$. We are using only $O(\sqrt{n})$ extra space for the tree and the heap.

3 2-d Proximity Problems

Theorem 2.3 implies a nearly in-place $O(n \log^3 n)$ algorithm for 2-d Voronoi diagrams 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.

3.1 Monochromatic and bichromatic closest pairs

Given a set P of n points in $\mathbb{R}^2$, there are a number of known algorithms to compute the closest pair in P in $O(n \log n)$ time. Some can be made in-place quite easily: The $O(n \log^2 n)$ version of the well-known divide-and-conquer algorithm [28] is one such example (in fact, this works in higher fixed dimensions); the $O(n \log n)$ version in 2-d can also be implemented without extra space but requires known techniques for in-place merging or stable partitioning. If coordinates are integers and the bitwise exclusive-or operation is supported, then there is a simpler solution (which also works in any fixed dimension): the "shift-shuffle-and-sort" algorithm suggested by Chan [4] is already in-place (using heapsort).

The bichromatic version of the problem is perhaps more interesting: given a set P of red points and a set Q of blue points in $\mathbb{R}^2$, we want $\min_{p,q \in P} \|p - q\|$. Here, we suggest Hinrichs et al.'s algorithm [18], which solves the following subproblem by a sweep: for each blue point q, find the nearest neighbor among all red points to the left of q (call this the *left nearest neighbor*). A symmetric subproblem (for right nearest neighbors) can be solved similarly, and we can take the minimum of the solutions found.

This approach is justified by the following intuition: While there is currently no "direct" sweep algorithm for the Voronoi diagram (without using divide-and-conquer, as we did in the previous section, or transforming the points, as Fortune [13] did), a direct sweep algorithm is nevertheless possible for the *left Voronoi diagram* (the planar subdivision where two points lie in the same cell if they have the same left nearest neighbors), because we know in advance the insertion time of each red point p (i.e., the smallest x-coordinate in its cell)—it is just the x-coordinate of p. It is not difficult to check that this approach can be implemented in $O(n \log^2 n)$ time with $O(\log^2 n)$ space, by the combined tree/heap data structure [10].

3.2 Bichromatic all-nearest-neighbors

For a more difficult example, suppose we want to compute the Hausdorff distance $\max_{q \in Q} \min_{p \in P} \|p - q\|$. We can adopt the same approach, but we face one difficulty: in performing the second sweep to compute the right nearest neighbors, we need to remember the left nearest neighbor to deduce the overall nearest neighbor of a point, but we don't have the space for it. This difficulty can be resolved by nontrivial modifications to the sweep:

Theorem 3.1 *Given a set P of n red points and a set Q of n blue points in R^2 stored in an array, 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.*

Proof: (Sketch) As described earlier, we can sweep from left to right over the left Voronoi diagram of the red points, but we can only record the deletion order resulting from this sweep. Let L be this deletion order.

We perform a second backward sweep, moving the sweep line ℓ from right to left. In this sweep, we maintain simultaneously a list V_L of red points whose left Voronoi cells intersect the sweep line ℓ, and a list V_R of red points whose right Voronoi cells intersect ℓ. These two lists are ordered from bottom to top. We can determine the next vertex to be inserted to V_L by removing the tail of the list L. On the other hand, a vertex is inserted to V_R when it is removed from V_L. The key observation for the space-efficient implementation is that the three lists L, V_L, and V_R are disjoint at all times. We store V_L and V_R in two separate combined tree/heap structures [10]. Whenever the sweep line hits a blue point q, we search the two trees to determine its left and right nearest neighbors and print the closer of the two. $\square$

4 Range Searching

In this section we demonstrate how certain known geometric data structures for range searching and related query problems can be made space-efficient.

4.1 An illustration

To illustrate the basic idea, we find it helpful to start with an early method for halfspace range counting in 2-d—Willard's partition tree [31]. Given a static set of n points in the plane, the preprocessing algorithm constructs two lines and recursively deals with the 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, 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 on these subarrays. The data structure requires no extra space! To answer a query, we only need to recurse on 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 and 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 [24]. We observe that Megiddo's ham-sandwich cut algorithm can actually be made in-place. This is similar to Brönnimann et al.'s observation [3] that Megiddo's 2-d linear programming algorithm can be made in-place. Details will appear in the long version. (Incidentally, Matoušek, Lo, and Steiger's algorithm [19] in the general 2-d non-separable case can also be made in-place in expected linear time.)

4.2 More implicit (static) 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 [20] for simplex range counting in an arbitrary fixed dimension d, which are obtained by the following 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 any hyperplane intersects at most $O(r^{1-1/d})$ simplices.

By inspecting Matoušek's construction closely, we can make several observations when we choose r to be a sufficiently large constant. First, the construction algorithm in the randomized version not only runs in $O(n)$ expected time but uses just a constant amount of extra space: the algorithm selects a random sample R of constant size and then works with a constant number of "test" hyperplanes defined by the sample. Second, the subsets $\{P_i\}$ actually have equal size if r divides n; by taking away a constant number of points S, we can therefore ensure that each $P_i \setminus (R \cup S)$ has exactly the same size. Third, the simplices are defined by (i.e., can be reconstructed from) R and a constant number of extra bits (which can be encoded by making $|S|$ sufficiently large and permuting pairs in S as in Section 2.1).

A space-efficient version of Matoušek's partition tree can now be obtained: store R and a permutation of S at the head of the array, divide the remaining array into r subarrays of equal size, and recursively store each subset $P_i \setminus (R \cup S)$. The data structure needs no extra space and can answer queries in $O(n^{1-1/d+\varepsilon})$ time and $O(1)$ space, for an ε that converges to 0 as $r \to \infty$.

A similar approach can be taken for Matoušek's halfspace range reporting structure [21], which in turn can be used to answer ray shooting and linear programming queries [22, 23]. (For linear programming queries, the method in [6] for example requires no changes to the tree structure.) Notice that nearest neighbor queries in $\mathbb{R}^d$ reduce to ray shooting in $\mathbb{R}^{d+1}$.

Theorem 4.1 *Given an array of n points in $\mathbb{R}^d$, we can permute the array in $O(n \log n)$ expected time, using $O(1)$ space, so that we can do one of the following:*

(i) *report all k points in any query simplex in $O(n^{1-1/d+\varepsilon} + k)$ time and $O(1)$ space;*

(ii) *report all k points in any query halfspace in $O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon} + k)$ time and $O(1)$ space;*

(iii) *perform ray shooting and linear programming queries in the dual polytope in $O(n^{1-1/\lfloor d/2 \rfloor+\varepsilon})$ time and $O(1)$ space.*

4.3 Implicit kd-trees

Even simpler than Willard's partition trees, kd-trees [11] can be constructed in-place in $O(n \log n)$ time, trivially. They can be used to answer orthogonal range queries in any fixed dimension d in time $O(n^{1-1/d} + k)$, and partial match queries (when $s < d$ coordinates are specified in a range) in time $O(n^{1-s/d} + k)$. Note that by Chazelle's lower bound [9], 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 is possible).

For three-sided orthogonal range searching in 2-d, however, the priority search tree [11] can be used to answer queries in $O(\log n + k)$ time and can be made in-place.

4.4 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 [1]. 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 increases 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 known reduction to range searching 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 [25]).

4.5 Applications to convex hulls

We point out one application to finding extreme points in higher dimensions.

Corollary 4.2 *Given an array of n points in $\mathbb{R}^d$, we can compute the vertices of the convex hull, stored in a prefix of the same array, in $O(n^{2-1/\lfloor d/2 \rfloor+\varepsilon})$ expected time using $O(1)$ extra space.*

Proof: Testing whether a point is extreme reduces to a linear programming query, so we can apply Theorem 4.1. One problem arises, though: we do not have extra space to record the answer for each point. To get around the problem, we modify the data structure to store also the answers implicitly within a permutation of S in each head block. Specifically, after testing all the points in a head block, we permute S so that all points with answer "yes" come before all points with answer "no". We then permute pairs in S as before, but recording the following additional pieces of information: which points of R have answer "yes" ($|R|$ bits), how many points of S have answer "yes" ($O(\log |S|)$ bits), and if this number is odd, whether the last point of S with answer "yes" has an even or odd index (1 bit). This is possible by making $|S|$ a sufficiently large constant, since $\log |S| \ll |S|$.

After all queries are answered, we apply a bottom-up postprocessing procedure to move all points with answer "yes" before all points with answer "no" in the array. $\square$

For another application of the in-place data structures, we derive an output-sensitive version of Theorem 2.3 for 3-d convex hulls, by modifying Chan's "group-and-wrap" algorithm [7]. Previously, Brönnimann et al. [3] considered space-efficient output-sensitive results for 2-d convex hulls.

Corollary 4.3 *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 $O(n \log^3 h)$ expected time using $O(\log^2 h)$ extra space. The edges and facets can also be printed within the same time bound.*

Proof: It suffices to find the extreme points, by Theorem 2.3. For simplicity, we assume that h is known; a standard "guessing" trick (e.g., [7]) can be applied otherwise. If $h \geq n^{1/4}$, then the bounds in Theorem 2.3 are good enough. So assume $h < n^{1/4}$. We divide the input array into one block of size $\Theta(h)$ and $O(n/h^4)$ blocks of size $O(h^4)$, and build a data structure for dual ray shooting queries for each block except the first, by Theorem 4.1. The total expected preprocessing time is $O(n \log h)$ 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. In each iteration, we go through each pair $p, q \in S$, test whether pq forms a hull edge, and if so, wraps the hull around $\overrightarrow{pq}$ 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 number of edge tests and wrapping queries is bounded by $O(h^3)$. These operations correspond to dual ray shooting queries on the whole input set, which require $O(h + (n/h^4)h^{4\varepsilon})$ time each by Theorem 4.1. In addition, $O(h)$ rebuilding operations are performed, each requiring $O(h \log h)$ expected time. The total running time after preprocessing can therefore be bounded by $O(n)$, since $h < n^{1/4}$. □

References

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

[2] 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*, 2004, to appear.

[3] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. In *Proc. Latin American Theoretical Informatics*, pages 494–507, 2002; *Theoret. Comput. Sci.*, to appear.

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

[5] 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.

[6] T. M. Chan. An optimal randomized algorithm for maximum Tukey depth. In *Proc. 15th ACM-SIAM Sympos. on Discrete Algorithms*, 2004, to appear.

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

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

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] C.-Y. Lo, J. Matoušek, and W. L. Steiger. Algorithms for ham-sandwich cuts. *Discrete Comput. Geom.*, 11:433–452, 1994.

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

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

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

[23] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete Comput. Geom.*, 10(2):215–232, 1993.

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

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

[27] 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.

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

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

[30] 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.

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

Proof of Lemma 2.2

We follow the notation in the kinetic 2-d setting. The data structure components consist of the following:

- Two input lists, containing the insertion order for LH(P_L) and for LH(P_R). For technical reasons, we actually need to allow the vertices to be slightly out of order:

  > Input Hypothesis: dividing each input list into pairs, we assume only that the insertion times of the vertices in a pair are smaller than the insertion times of the vertices in the next pair.

As time progresses, pairs will be removed from the input lists.

- An output list, containing the vertices that have been deleted from LH(P) so far, in deletion order. A point is said to be *live* if it has not yet been put in the output list.
- Two additional lists $Q_L \subseteq P_L$ and $Q_R \subseteq P_R$ (in left-to-right order), where at any time,

  Q_L INVARIANTS: 1. Q_L contains all live vertices of the current hull LH(P_L) (and possibly additional points), and 2. all points of Q_L are vertices of LH(Q_L), except for at most two points (called the *interior points*),

  and Q_R obeys similar properties.

At any time, a point belongs to at most one of the above lists. The lists supposedly all reside in the given array, although the algorithm is best explained first without consideration of the memory layout:

The Algorithm. Let v^+ (resp. v^-) denote the successor (resp. predecessor) vertex of a given vertex v of LH(Q_L) or LH(Q_R); for a non-vertex v, let v^+ (resp. v^-) denote the vertices immediately to the right (resp. left) of v in LH(Q_L) or LH(Q_R). The pair $b_L b_R$ denotes the current bridge of the two hulls LH(P_L) and LH(P_R).

The algorithm works iteratively. Initialization and termination details are again omitted. At each iteration, we identify the first pairs π_L and π_R of the two input lists and generate the following set of possible events, each specified by a time value and an accompanying action:

- EVENT 0.
  Time: the first time when $q^- q q^+$ becomes concave for some vertex q of LH(Q_L).
  Action: delete this q from Q_L and update LH(Q_L); if q is left of b_L, put q in the output list.
- EVENT 1, to be considered if there are interior points in Q_L.
  Time: the first time when $q^- q q^+$ becomes convex for some interior point q.
  Action: update LH(Q_L).
- EVENT 2, to be considered if pp^+p^{++} will never be convex for some $p \in \pi_L$.
  Time: now. Action: delete p^+ from Q_L.
- EVENT 3, to be considered if pp^+p^{++} is concave but will eventually be convex for some $p \in \pi_L$.
  Time: when pp^+p^{++} becomes convex. Action: none.
- EVENTS 2–3, but with signs ($+$ and $-$) switched.
- EVENT 4, to be considered if there is no interior point in Q_L and pp^-p^{--} and pp^+p^{++} are both convex for all $p \in \pi_L$.
  Time: now. Action: insert π_L to Q_L, update LH(Q_L), and remove π_L from its input list.
- All of the above events, but with sides (left and right, L and R) switched.
- EVENT 5.
  Time: when $b_L b_L^+ b_R$ is convex. Action: $b_L \leftarrow b_L^+$.
- EVENT 6.
  Time: when $b_L^- b_L b_R$ is concave.
  Action: delete b_L from Q_L, update LH(Q_L), and put b_L in the output list; $b_L \leftarrow b_L^-$.
- EVENTS 5–6, but with signs and sides simultaneously switched.

From these constant number of possible events, we take the one with the smallest time value, advance the current time to this value, execute the corresponding action, and continue with the next iteration (with a new set of events).

Correctness. To prove that the algorithm is correct, we need to verify carefully all the stated invariants. We provide below a sample of justifications for some of the actions:

- EVENT 0. If q^-qq^+ just becomes concave, q will never be a vertex of LH(P_L) and can be safely deleted from Q_L. Furthermore, LH(Q_L) coincides with LH(P_L) to the left of b_L, because all vertices of LH(P_L) to the left of b_L are live. So, if q is left of b_L, then q is being deleted from LH(P).
- EVENT 2. If pp^+p^{++} will never be convex, it will never be a vertex of LH(P_L) and can be safely deleted from Q_L.
- EVENT 4. If pp^-p^{--} and pp^+p^{++} are both convex for each $p \in \pi_L$, inserting π_L to Q_L creates at most two additional interior points.
- EVENT 5. The bridge is correctly maintained here, because the vertices of the bridge of LH(P_L) and LH(P_R) are always live and thus in LH(Q_L) and LH(Q_R).
- EVENT 6. Here, b_L is dead and can be deleted from Q_L.

To verify the first of the Q_L invariants, we also need to show that each point $v \in \pi_L$ is inserted to Q_L before it becomes a vertex in LH(P_L). Let $t[v]$ denote the insertion time of v in LH(P_L). At any time, at least one of Events 1–4 is applicable. If Event 1 is executed, time is advanced to a value $\leq t[q]$, since q^-qq^+ must be convex by time $t[q]$; by the input hypothesis, $t[q] \leq t[v]$. If Event 3 is executed, time is advanced to a value $\leq t[p^+]$, which is again $\leq t[v]$. Thus, the current time is always $\leq t[v]$ until π_L is inserted to Q_L (by Event 4).

Note that the number of iterations is $O(n)$: each iteration can be charged to an insertion/deletion in Q_L or Q_R, or a change in LH(P); the only exceptions appear to be when Event 1 or 3 is executed, but unless an insertion/deletion in Q_L or Q_R occurs, there can be at most $O(1)$ executions of such events.

Space-Efficient Implementation. To implement the algorithm, we partition the given array and the extra space into blocks of size $s = 2c \log n$, where each block is used for one of the lists (or as garbage). Each input list is stored as an implicit doubly-linked list; this requires us to permute certain pairs of points in the list, but our weaker input hypothesis allows precisely for this. Each of the lists Q_L and Q_R is stored as an implicit search tree layered with a heap, as in the previous proof, where the priority value of a vertex q is the time when q^-qq^+ becomes concave.

The output list is stored in a prefix of the given array. Whenever the output list expands to a new block, we swap the block with a free block. Swapping blocks requires adjusting a constant number of pointers in one of the doubly linked lists or implicit search trees and takes $O(1)$ amortized time ($O(s)$ time for every s operations). Whenever a free block is generated from the input list or the implicit search tree, we similarly swap the block with the last used block.

All blocks are full, except for one block for each doubly linked list, $O(\log n)$ blocks for each implicit search tree [10, 25], and one garbage block. So, the amount of extra space is bounded by $O(\log^2 n)$. Each operation on Q_L or Q_R takes $O(\log^2 n)$ time [10, 25]. So, the algorithm runs in $O(n \log^2 n)$ time. □