Time-Space Trade-offs for Triangulations and Voronoi Diagrams

Matias Korman[1], Wolfgang Mulzer[2⋆], André van Renssen[1], Marcel Roeloffzen[3], Paul Seiferth[2⋆], and Yannik Stein[2⋆]

[1] National Institute of Informatics (NII), Tokyo, Japan;
JST, ERATO, Kawarabayashi Large Graph Project. {korman,andre}@nii.ac.jp
[2] Institut für Informatik, Freie Universität Berlin, Germany.
{mulzer,pseiferth,yannikstein}@inf.fu-berlin.de
[3] Tohoku University, Japan. marcel@dais.is.tohoku.ac.jp

Abstract. Let S be a planar n-point set. A *triangulation* for S is a maximal plane straight-line graph with vertex set S. The *Voronoi diagram* for S is the subdivision of the plane into cells such that each cell has the same nearest neighbors in S. Classically, both structures can be computed in $O(n \log n)$ time and $O(n)$ space. We study the situation when the available workspace is limited: given a parameter $s \in \{1, \ldots, n\}$, an s-workspace algorithm has read-only access to an input array with the points from S in arbitrary order, and it may use only $O(s)$ additional words of $\Theta(\log n)$ bits for reading and writing intermediate data. The output should then be written to a write-only structure. We describe a deterministic s-workspace algorithm for computing a triangulation of S in time $O(n^2/s + n \log n \log s)$ and a randomized s-workspace algorithm for finding the Voronoi diagram of S in expected time $O((n^2/s) \log s + n \log s \log^* s)$.

1 Introduction

Since the early days of computer science, a major concern has been to cope with strong memory constraints. This started in the '70s [21] when memory was expensive. Nowadays, the motivation comes from a proliferation of small embedded devices where large memory is neither feasible nor desirable (e.g., due to constraints on budget, power, size, or simply to discourage potential thievery).

Even when memory size is not an issue, we might want to limit the number of write operations: one can read flash memory quickly, but writing (or even reordering) data is slow and may reduce the lifetime of the storage system; write-access to removable memory may be limited for technical or security reasons, (e.g., when using read-only media such as DVDs or to prevent leaking information about the algorithm). Similar problems occur when concurrent algorithms access data simultaneously. A natural way to address this is to consider algorithms that do not modify the input.

⋆ WS and PS were supported in part by DFG Grants MU 3501/1 and MU 3501/2. YS was supported by the DFG within the research training group MDS (GRK 1408).

The exact setting may vary, but there is a common theme: the input resides in read-only memory, the output must be written to a write-only structure, and we can use $O(s)$ additional variables to find the solution (for a parameter s). The goal is to design algorithms whose running time decreases as s grows, giving a *time-space trade-off* [22]. One of the first problems considered in this model is *sorting* [18,19]. Here, the time-space product is known to be $\Omega(n^2)$ [8], and matching upper bounds for the case $b \in \Omega(\log n) \cap O(n/\log n)$ were obtained by Pagter and Rauhe [20] (b denotes the available workspace in *bits*).

Our current notion of memory constrained algorithms was introduced to computational geometry by Asano *et al.* [4], who show how to compute many classic geometric structures with $O(1)$ workspace (related models were studied before [9]). Later, time-space trade-offs were given for problems on simple polygons, e.g., shortest paths [1], visibility [6], or the convex hull of the vertices [5].

In our model, we are given an array S of n points in the plane such that random access to each input point is possible, but we may not change or even reorder the input. Additionally, we have $O(s)$ variables (for a parameter $s \in \{1, \ldots, n\}$). We assume that each variable or pointer contains a data word of $\Theta(\log n)$ bits. Other than this, the model allows the usual word RAM operations. We consider two problems: computing an arbitrary triangulation for S and computing the Voronoi diagram $\mathrm{VD}(S)$ for S. Since the output cannot be stored explicitly, the goal is to report the edges of the triangulation or the vertices of $\mathrm{VD}(S)$ successively, in no particular order. Dually, the latter goal may be phrased in terms of Delaunay triangulations. We focus on Voronoi diagrams, as they lead to a more natural presentation.

Both problems can be solved in $O(n^2)$ time with $O(1)$ workspace [4] or in $O(n \log n)$ time with $O(n)$ workspace [7]. However, to the best of our knowledge, no trade-offs were known before. Our triangulation algorithm achieves a running time of $O(n^2/s + n \log n \log s)$ using $O(s)$ variables. A key ingredient is the recent time-space trade-off by Asano and Kirkpatrick for a special type of simple polygons [3]. This also lets us obtain significantly better running times for the case that the input is sorted in x-order; see Section 2. For Voronoi diagrams, we use random sampling to find the result in expected time $O((n^2 \log s)/s + n \log s \log^* s))$; see Section 3. Together with recent work of Har-Peled [15], this appears to be one of the first uses of random sampling to obtain space-time trade-offs for geometric algorithms. The sorting lower bounds also apply to triangulations (since we can reduce the former to the latter). By duality between Voronoi diagrams and Delaunay triangulations this implies that our second algorithm for computing the Voronoi diagram is almost optimal.

2 Triangulating a Sorted Point Set

We first describe our s-workspace algorithm for triangulating a planar point set that is given in sorted x-order. The input points $S = \{q_1, \ldots, q_n\}$ are stored by increasing x-coordinate, and we assume that all x-coordinates are distinct, i.e., $x_i < x_{i+1}$ for $1 \leq i < n$, where x_i denotes the x-coordinate of q_i, for $1 \leq i \leq n$.

A crucial ingredient in our algorithms is a recent result by Asano and Kirkpatrick for triangulating *monotone mountains*[1] (or *mountains* for short). A mountain is a simple polygon with vertex sequence $v_1, v_2, \ldots, v_k$ such that the x-coordinates of the vertices increase monotonically. The edge v_1v_k is called the *base*. Mountains can be triangulated very efficiently with bounded workspace.

Theorem 2.1 (Lemma 3 in [3], rephrased). *Let H be a mountain with n vertices, stored in sorted x-order in read-only memory. Let $s \in \{2, \ldots, n\}$. We can report the edges of a triangulation of H in $O(n \log_s n)$ time and $O(s)$ space.*

Since S is given in x-order, the edges q_iq_{i+1}, for $1 \leq i < n$, form a monotone simple polygonal chain. Let $\mathrm{Part}(S)$ be the subdivision obtained by the union of this chain with the edges of the convex hull of S. We say that a convex hull edge is *long* if the difference between its indices is at least two (i.e., the endpoints are not consecutive). The following lemma lets us decompose the problem into smaller pieces. The proof can be found in the full version.

Lemma 2.2. *Any bounded face of* $\mathrm{Part}(S)$ *is a mountain whose base is a long convex hull edge. Moreover, no point of S lies in more than four faces of* $\mathrm{Part}(S)$.

Let $e_1, \ldots, e_k$ be the long edges of the convex hull of S, let F_i be the unique mountain of $\mathrm{Part}(S)$ whose base is e_i, and let n_i be the number of vertices of F_i.

With the above definitions we can give an overview of our algorithm. We start by computing the edges of the upper convex hull, from left to right. Each time an edge e_i of the convex hull is found, we check if it is long. If so, we triangulate the corresponding mountain F_i (otherwise we do nothing), and we proceed with the computation of the convex hull. The algorithm finishes once all convex hull edges are computed (and their corresponding faces have been triangulated).

Theorem 2.3. *Let S be a set of n points, sorted in x-order. We can report the edges of a triangulation of S in $O(n^2)$ time using $O(1)$ variables, $O(n^2 \log n/2^s)$ time using $O(s)$ additional variables (for any $s \in \Omega(\log\log n) \cap o(\log n)$), or $O(n \log_p n)$ time using $O(p \log_p n)$ additional variables (for any $2 \leq p \leq n$).*

Proof. Correctness follows directly from the first claim of Lemma 2.2. Thus, it suffices to show the performance bounds. The main steps are: (i) computing the convex hull of a point set given in x-order; (ii) determining if an edge is long; and (iii) triangulating a mountain. We can identify long edges in constant time, by comparing the endpoint indices. Moreover, by Theorem 2.1, we can triangulate the polygon F_i of n_i vertices in time $O(n_i \log_s n_i)$ with $O(s)$ variables.

Further note that we never need to store the polygons to triangulate explicitly: once a long edge e_i of the convex hull is found, we can implicitly give it as input for the triangulation algorithm by specifying the indices of e_i: since the points are sorted by x-coordinate, the other vertices of F_i are exactly the input points between the endpoints of e_i. We then pause our convex-hull computation, keeping its current state in memory, and reserve $O(s)$ memory for triangulating

[1] Also known as *unimonotone polygon* [14].

F_i. Once this is done, we can discard all used memory, and we reuse that space for the next mountain. We then continue the convex hull algorithm to find the next long convex-hull edge. By the second claim of Lemma 2.2, no vertex appears in more than four mountains, and thus the total time for triangulating all mountains is bounded by $\sum_i O(n_i \log_s n_i) = O(n \log_s n)$.

Finally, we need to bound the time for computing the convex hull of S. Recall that we may temporarily pause the algorithm to triangulate a generated polygon, but overall it is executed only once over the input. There exist several algorithms for computing the convex hull of a set of points sorted by x-coordinate under memory constraints. When $s \in \Theta(1)$, we can use the gift-wrapping algorithm (Jarvis march [16]) which runs in $O(n^2)$ time. Barba *et al.* [5] provided a different algorithm that runs in $O(n^2 \log n/2^s)$ time using $O(s)$ variables (for any $s \in o(\log n)$).[2] This approach is desirable for $s \in \Omega(\log \log n) \cap o(\log n)$. As soon as the workspace can fit $\Omega(\log n)$ variables, we can use the approach of Chan and Chen [10]. This algorithm runs in $O(n \log_p n)$ time and uses $O(p \log_p n)$ variables, for any $2 \leq p \leq n$. In any case, the time for the convex hull dominates the time for triangulating the mountain mountains. □

General Input. The previous algorithm uses the sorted input order in two ways. Firstly, the algorithms of Barba *et al.* [5] and of Chan and Chen [10] work only for simple polygons (e.g., for sorted input). Instead, we may use the algorithm by Darwish and Elmasry [13] that gives the upper (or lower) convex hull of any sequence of n points in $O(n^2/(s \log n) + n \log n)$ time with $O(s)$ variables[3], matching known lower bounds. Secondly, and more importantly, the Asano-Kirkpatrick (AK) algorithm requires the input to be sorted. To address this issue, we simulate sorted input using multiple heap structures. For this, we require some technical details on how the AK-algorithm accesses its input.

Let F be a mountain with n vertices. Let $F^{\uparrow}$ denote the vertices of F in ascending x-order, and $F^{\downarrow}$ denote F in descending x-order. The AK-algorithm makes one pass over $F^{\uparrow}$ and one pass over $F^{\downarrow}$.[4] Each pass computes half of the triangulation, uses $O(s)$ variables and has $\Theta(\log_s n)$ rounds. In round i, it partitions $F^{\uparrow}$ ($F^{\downarrow}$) into blocks of $O(|F|/s^i)$ consecutive points that are processed from left to right. Each block is further subdivided into $O(s)$ sub-blocks $b_1, \ldots, b_k$ of size $O(|F|/s^{i+1})$. The algorithm does two scans over the sub-blocks. The first scan processes the elements in x-order. Whenever the first scan finishes reading a sub-block b_i, the algorithm makes b_i active and creates a pointer l_i to the rightmost element of b_i. The second scan goes from right to left and is concurrent to the first scan. In each step, it reads the element at l_i in the rightmost active

[2] In fact, Barba *et al.* show how to compute the convex hull of a simple polygon, but also show that both problems are equivalent. The monotone chain can be completed to a polygon by adding a vertex with a very high or low y-coordinate.

[3] Darwish and Elmasry [13] state a running time of $O(n^2/s+n \log n)$, but they measure workspace in bits while we use words.

[4] AK reduce triangulation to the *all next smaller right neighbor* (NSR) and the *all next smaller left neighbor* (NSL) problem and present an algorithm for NSR if the input is in x-order. This implies an NSL-algorithm by reading the input in reverse.

sub-block b_i, and it decreases l_i by one. If l_i leaves b_i, then b_i becomes inactive. As the first scan creates new active sub-blocks as it proceeds, the second scan may jump between sub-blocks.

We use the heap by Asano *et al.* [2] to provide the input for the AK-algorithm. We shortly restate its properties.

Lemma 2.4 ([2]). *Let S be a set of n points. There is a heap that supports insert and extract-min (resp. extract-max) in $O\big((n/(s\log n)+\log s)D(n)\big)$ time using $O(s)$ variables, where $D(n)$ is the time to decide whether a given element currently resides in the heap (is* alive*).*[5]

Lemma 2.5 ([2]). *Let S be a set of n points. We can build a heap with all elements in S in $O(n)$ time that supports extract-min in $O\big(n/(s\log n)+\log(n)\big)$ time using $O(s)$ variables.*

Proof. The construction time is given in [2]. To decide in $O(1)$ time if some $x \in S$ is alive, we store the last extracted minimum m and test whether $x > m$. □

We now present the complete algorithm. We first show how to subdivide S into mountains F_i and how to run the AK-algorithm on each $F_i^{\uparrow}$. Finally, we discuss the changes to run the AK-algorithm on each $F_i^{\downarrow}$. Sorted input is emulated by constructing two heaps H_1,H_2 for S according to x-order. By Lemma 2.5, each heap uses $O(s)$ space, can be constructed in $O(n)$ time, and supports extract-min in $O(n/(s\log n)+\log n)$ worst-case time. With H_1 we determine the size of the next mountain F_i, with H_2 we process the points of F_i.

We execute the convex hull algorithm with $\Theta(s)$ space until it reports the next convex hull edge pq. Throughout, the heaps H_1 and H_2 contain exactly the points to the right of p. To determine if pq is long (i.e., if there are points between p and q), we repeatedly extract the minimum of H_1 until q becomes the minimum element. Let k be the number of removed points.

If $k = 1$, then pq is short. We extract the minimum of H_2, and we continue with the convex hull algorithm. If $k \geq 2$, Lemma 2.2 shows that pq is the base of a mountain F that consists of all points between p and q. These are exactly the $k+1$ smallest elements in H_2 (including p and q). If $k \leq s$, we extract them from H_2, and we triangulate F in memory. If $k > s$, we execute the AK-algorithm on F using $O(s)$ variables. At the beginning of the ith round, we create a copy $H_{(i)}$ of H_2, i.e., we duplicate the $O(s)$ variables that determine the state of H_2. Further, we create an empty max-heap $H_{(ii)}$ using $O(s)$ variables to provide input for the second scan. To be able to reread a sub-block, we create a further copy $H'_{(i)}$ of H_2. Whenever the AK-algorithm requests the next point p in the first scan, we simply extract the minimum of $H_{(i)}$. When a sub-block is fully read, we use $H'_{(i)}$ to reread the elements and insert them into $H_{(ii)}$. Now, the rightmost element of all active sub-blocks corresponds exactly to the maximum of $H_{(ii)}$. One step in the second scan is equivalent to an extract-max on $H_{(ii)}$.

[5] The bounds in [2] do not include the factor $D(n)$ since the authors studied a setting similar to Lemma 2.5 where it takes $O(1)$ time to decide whether an element is alive.

At the end of one round, we delete $H_{(i)}$, $H'_{(i)}$, and $H_{(ii)}$. The space can be reused in the next round. Once the AK-algorithm finishes, we repeatedly extract the minimum of H_2 until the minimum becomes q.

For each mountain F with $|F| > s$, the algorithm runs the AK-algorithm on $F^{\uparrow}$. To output a complete triangulation of F, we repeat the whole algorithm on S in reverse order, so that the AK-algorithm is run on each $F^{\downarrow}$.

Theorem 2.6. *We can report the edges of a triangulation of a set S of n points in time $O(n^2/s + n \log n \log s)$ using $O(s)$ additional variables.*

Proof. As before, correctness directly follows from Lemma 2.2 and the correctness of the AK-algorithm. The bound on the space usage is immediate.

Computing the convex hull now needs $O(n^2/(s \log n) + n \log n)$ time [13]. By Lemma 2.5, the heaps H_1 and H_2 can be constructed in $O(n)$ time. During execution, we perform n extract-min operations on each heap, requiring $O(n^2/(s \log n) + n \log n)$ time in total.

Let F_j be a mountain with n_j vertices that is discovered by the convex hull algorithm. If $n_j \leq s$, then F_j is triangulated in memory in $O(n_j)$ time, and the total time for such mountains is $O(n)$. If $n_j > s$, then the AK-algorithm runs in $O(n_j \log_s n_j)$ time. We must also account for providing the input for the algorithm. For this, consider some round $i \geq 1$. We copy H_2 to $H_{(i)}$ in $O(s)$ time. This time can be charged to the first scan, since $n_j > s$. Furthermore, we perform n_j extract-min operations on $H_{(i)}$. Hence the total time to provide input for the first scan is $O(n_j n/(s \log n) + n_j \log n)$.

For the second scan, we create another copy $H'_{(i)}$ of H_2. Again, the time for this can be charged to the scan. Also, we perform n_j extract-min operations on $H'_{(i)}$ which takes $O(n_j n/(s \log n) + n_j \log n)$ time. Additionally, we insert each fully-read block into $H_{(ii)}$. The main problem is to determine if an element in $H_{(ii)}$ is alive: there are at most $O(s)$ active sub-blocks. For each active sub-block b_i, we know the first element y_i and the element z_i that l_i points to. An element is alive if and only if it is in the interval $[y_i, z_i]$ for some active b_i. This can be checked in $O(\log s)$ time. Thus, by Lemma 2.4, each insert and extract-max on $H_{(ii)}$ takes $O\big((n/(s \log n) + \log s) \log s\big)$ time. Since each element is inserted once, the total time to provide input to the second scan is $O(n_j \log(s)(n/(s \log n) + \log s))$. This dominates the time for the first scan. There are $O(\log_s n_j)$ rounds, so we can triangulate F_j in time $O\big(n_j \log_s n_j + n_j \log(n_j)\big(n/(s \log n) + \log s\big)\big)$. Summing over all F_j, the total time is $O(n^2/s + n \log n \log s)$. □

3 Voronoi Diagrams

Given a planar n-point set S, we would like to find the vertices of $\mathrm{VD}(S)$. Let $K = \{p_1, p_2, p_3\}$ be a triangle with $S \subseteq \mathrm{conv}(K)$ so that all vertices of $\mathrm{VD}(S)$ are vertices of $\mathrm{VD}(S \cup K)$. We use random sampling to divide the problem of computing $\mathrm{VD}(S \cup K)$ into $O(s)$ subproblems of size $O(n/s)$. First, we show how to take a random sample from S with small workspace. One of many possible approaches is the following deterministic one that ensures a worst-case guarantee:

Lemma 3.1. *We can sample a uniform random subset $R \subseteq S$ of size s in time $O(n + s \log s)$ and space $O(s)$.*

Proof. We sample a random sequence I of s distinct numbers from $\{1, \ldots, n\}$. This is done in s *rounds*. At the beginning of round k, for $k = 1, \ldots, s$, we have a sequence I of $k-1$ numbers from $\{1, \ldots, n\}$. We store I in a binary search tree T. We maintain the invariant that each node in T with value in $\{1, \ldots, n-k+1\}$ stores a pointer to a unique number in $\{n-k+2, \ldots, n\}$ that is not in I. In round k, we sample a random number x from $\{1, \ldots, n-k+1\}$, and we check in T whether $x \in I$. If not, we add x to I. Otherwise, we add to I the number that x points to. Let y be the new element. We add y to T. Then we update the pointers: if $x = n-k+1$, we do nothing. Now suppose $x < n-k+1$. Then, if $n-k+1 \notin I$, we put a pointer from x to $n-k+1$. Otherwise, if $n-k+1 \in I$, we let x point to the element that $n-k+1$ points to. This keeps the invariant and takes $O(\log s)$ time and $O(s)$ space. We continue for s rounds. Any sequence of s distinct numbers in $\{1, \ldots, n\}$ is sampled with equal probability.

Finally, we scan through S to obtain the elements whose positions correspond to the numbers in I. This requires $O(n)$ time and $O(s)$ space.

□

We use Lemma 3.1 to find a random sample $R \subseteq S$ of size s. We compute $\mathrm{VD}(R \cup K)$, triangulate the bounded cells and construct a planar point location structure for the triangulation. This takes $O(s \log s)$ time and $O(s)$ space [17]. Given a vertex $v \in \mathrm{VD}(R \cup K)$, the *conflict circle* of v is the largest circle with center v and no point from $R \cup K$ in its interior. The *conflict set* B_v of v contains all points from S that lie in the conflict circle of v, and the *conflict size* b_v of v is $|B_v|$. We scan through S to find the conflict size b_v for each vertex $v \in \mathrm{VD}(R \cup K)$: every Voronoi vertex has a counter that is initially 0. For each $p \in S \setminus (R \cup K)$, we use the point location structure to find the triangle Δ of $\mathrm{VD}(R \cup K)$ that contains it. At least one vertex v of Δ is in conflict with p. Starting from v, we walk along the edges of $\mathrm{VD}(R \cup K)$ to find all Voronoi vertices in conflict with p. We increment the counters of all these vertices. This may take a long time in the worst case, so we impose an upper bound on the total work. For this, we choose a *threshold* M. When the sum of the conflict counters exceeds M, we start over with a new sample R. The total time for one attempt is $O(n \log s + M)$, and below we prove that for $M = \Theta(n)$ the success probability is at least $3/4$. Next, we pick another threshold T, and we compute for each vertex v of $\mathrm{VD}(R \cup K)$ the *excess* $t_v = b_v s/n$. The excess measures how far the vertex deviates from the desired conflict size n/s. We check if $\sum_{v \in \mathrm{VD}(R \cup K)} t_v \log t_v \leq T$. If not, we start over with a new sample. Below, we prove that for $T = \Theta(s)$, the success probability is at least $3/4$. The total success probability is $1/2$, and the expected number of attempts is 2. Thus, in expected time $O(n \log s + s \log s)$, we can find a sample $R \subseteq S$ with $\sum_{v \in \mathrm{VD}(R \cup K)} b_v = O(n)$ and $\sum_{v \in \mathrm{VD}(R \cup K)} t_v \log t_v = O(s)$.

We now analyze the success probabilities, using the classic Clarkson-Shor method [12]. We begin with a variant of the Chazelle-Friedman bound [11].

Lemma 3.2. *Let X be a planar point set of size o, and let $Y \subset \mathbb{R}^2$ with $|Y| \leq 3$. For fixed $p \in (0,1]$, let $R \subseteq X$ be a random subset of size po and let $R' \subseteq X$ be a random subset of size $p'o$, for $p' = p/2$. Suppose that $p'o \geq 4$. Fix $\mathbf{u} \in X^3$, and let $v_{\mathbf{u}}$ be the Voronoi vertex defined by $\mathbf{u}$. Let $b_{\mathbf{u}}$ be the number of points from X in the largest circle with center $v_{\mathbf{u}}$ and with no points from R in its interior. Then,*

$$\Pr[v_{\mathbf{u}} \in \mathrm{VD}(R \cup Y)] \leq 64 e^{-pb_{\mathbf{u}}/2} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R' \cup Y)].$$

Proof. Let $\sigma = \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R \cup Y)]$ and $\sigma' = \Pr[v_{\mathbf{u}} \in \mathrm{DT}(R' \cup Y)]$. The vertex $v_{\mathbf{u}}$ is in $\mathrm{VD}(R \cup Y)$ precisely if $\mathbf{u} \subseteq R \cup Y$ and $B_{\mathbf{u}} \cap (R \cup Y) = \emptyset$, where $B_{\mathbf{u}}$ are the points from X in the conflict circle of $v_{\mathbf{u}}$. If $Y \cap B_{\mathbf{u}} \neq \emptyset$, then $\sigma = \sigma' = 0$, and the lemma holds. Thus, assume that $Y \cap B_{\mathbf{u}} = \emptyset$. Let $d_{\mathbf{u}} = |\mathbf{u} \setminus Y|$, the number of points in $\mathbf{u}$ not in Y. There are $\binom{o-b_{\mathbf{u}}-d_{\mathbf{u}}}{po-d_{\mathbf{u}}}$ ways to choose a po-subset from X that avoids all points in $B_{\mathbf{u}}$ and contains all points of $\mathbf{u} \cap X$, so

$$\sigma = \binom{o-b_{\mathbf{u}}-d_{\mathbf{u}}}{po-d_{\mathbf{u}}} \Big/ \binom{o}{po} = \frac{\prod_{j=0}^{po-d_{\mathbf{u}}-1}(o-b_{\mathbf{u}}-d_{\mathbf{u}}-j)}{\prod_{j=0}^{po-d_{\mathbf{u}}-1}(po-d_{\mathbf{u}}-j)} \Big/ \frac{\prod_{j=0}^{po-1}(o-j)}{\prod_{j=0}^{po-1}(po-j)}$$
$$= \prod_{j=0}^{d_{\mathbf{u}}-1} \frac{po-j}{o-j} \cdot \prod_{j=0}^{po-d_{\mathbf{u}}-1} \frac{o-b_{\mathbf{u}}-d_{\mathbf{u}}-j}{o-d_{\mathbf{u}}-j} \leq p^{d_{\mathbf{u}}} \prod_{j=0}^{po-d_{\mathbf{u}}-1} \left(1 - \frac{b_{\mathbf{u}}}{o-d_{\mathbf{u}}-j}\right).$$

Similarly, we get

$$\sigma' = \prod_{i=0}^{d_{\mathbf{u}}-1} \frac{p'o-i}{o-i} \prod_{j=0}^{p'o-d_{\mathbf{u}}-1} \left(1 - \frac{b_{\mathbf{u}}}{o-d_{\mathbf{u}}-j}\right),$$

and since $p'o \geq 4$ and $i \leq 2$, it follows that

$$\sigma' \geq \left(\frac{p'}{2}\right)^{d_{\mathbf{u}}} \prod_{j=0}^{p'o-d_{\mathbf{u}}-1} \left(1 - \frac{b_{\mathbf{u}}}{o-d_{\mathbf{u}}-j}\right).$$

Therefore, since $p' = p/2$,

$$\frac{\sigma}{\sigma'} \leq \left(\frac{2p}{p'}\right)^{d_{\mathbf{u}}} \prod_{j=p'o-d_{\mathbf{u}}}^{po-d_{\mathbf{u}}-1} \left(1 - \frac{b_{\mathbf{u}}}{o-d_{\mathbf{u}}-j}\right) \leq 64 \left(1 - \frac{b_{\mathbf{u}}}{o}\right)^{po/2} \leq 64\, e^{pb_{\mathbf{u}}/2}.$$

□

We can now bound the total expected conflict size.

Lemma 3.3. *We have* $\mathbf{E}\left[\sum_{v \in \mathrm{VD}(R \cup K)} b_v\right] = O(n)$.

Proof. By expanding the expectation, we get

$$\mathbf{E}\left[\sum_{v \in \mathrm{VD}(R \cup K)} b_v\right] = \sum_{\mathbf{u} \in S^3} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R \cup K)] b_{\mathbf{u}},$$

$v_{\mathbf{u}}$ being the Voronoi vertex of $\mathbf{u}$ and $b_{\mathbf{u}}$ its conflict size. By Lemma 3.2 with $X = S$, $Y = K$ and $p = s/n$,

$$\leq \sum_{\mathbf{u} \in S^3} 64 e^{-p b_{\mathbf{u}}/2} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R' \cup K)] b_{\mathbf{u}},$$

where $R' \subseteq S$ is a sample of size $s/2$. We estimate

$$\begin{aligned}
&\leq \sum_{t=0}^{\infty} \sum_{\substack{\mathbf{u} \in S^3 \\ b_{\mathbf{u}} \in [\frac{t}{p}, \frac{t+1}{p})}} \frac{64 e^{-t/2}(t+1)}{p} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R' \cup K)] \\
&\leq \frac{1}{p} \sum_{\mathbf{u} \in S^3} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R' \cup K)] \sum_{t=0}^{\infty} 64 e^{-t/2}(t+1) \\
&= O(s/p) = O(n),
\end{aligned}$$

since $\sum_{\mathbf{u} \in S^3} \Pr[v_{\mathbf{u}} \in \mathrm{VD}(R' \cup K)] = O(s)$ is the size of $\mathrm{VD}(R' \cup K)$ and $\sum_{t=0}^{\infty} e^{-t/2}(t+1) = O(1)$. □

By Lemma 3.3 and Markov's inequality, it follows that there is an $M = \Theta(n)$ with $\Pr[\sum_{v \in \mathrm{VD}(R \cup K)} b_v > M] \leq 1/4$. The proof for the excess is very similar to the previous calculation and can be found in the full version.

Lemma 3.4. $\mathbf{E}\left[\sum_{v \in \mathrm{VD}(R \cup K)} t_v \log t_v\right] = O(s)$.

By Markov's inequality and Lemma 3.4, we can conclude that there is a $T = \Theta(s)$ with $\Pr[\sum_{v \in \mathrm{VD}(R \cup K)} t_v \log t_v \geq T] \leq 1/4$. This finishes the first sampling phase. The next goal is to sample for each vertex v with $t_v \geq 2$ a random subset $R_v \subseteq B_v$ of size $\alpha t_v \log t_v$ for large enough $\alpha > 0$ (recall that B_v is the conflict set of v).

Lemma 3.5. *In total time $O(n \log s)$, we can sample for each vertex $v \in \mathrm{VD}(R \cup K)$ with $t_v \geq 2$ a random subset $R_v \subseteq B_v$ of size $\alpha t_v \log t_v$.*

Proof. First, we perform $O(s)$ rounds to sample for each vertex v with $t_v \geq 2$ a sequence I_v of $\alpha t_v \log t_v$ distinct numbers from $\{1, \ldots, b_v\}$. For this, we use the algorithm from Lemma 3.1 in parallel for each relevant vertex from $\mathrm{VD}(R \cup K)$. Since $\sum_v t_v \log t_v = O(s)$, this takes total time $O(s \log s)$ and total space $O(s)$.

After that, we scan through S. For each vertex v, we have a counter c_v, initialized to 0. For each $p \in S$, we find the conflict vertices of p, and for each conflict vertex v, we increment c_v. If c_v appears in the corresponding set I_v, we add p to R_v. The total running time is $O(n \log s)$, as we do one point location for each input point and the total conflict size is $O(n)$.

□

We next show that for a *fixed* vertex $v \in \mathrm{VD}(R \cup K)$, with constant probability, all vertices in $\mathrm{VD}(R_v)$ have conflict size n/s with respect to B_v.

Lemma 3.6. *Let $v \in \mathrm{VD}(R \cup K)$ with $t_v \geq 2$, and let $R_v \subseteq B_v$ be the sample for v. The expected number of vertices v' in $\mathrm{VD}(R_v)$ with at least n/s points from B_v in their conflict circle is at most $1/4$.*

Proof. Recall that $t_v = b_v s/n$. We have

$$\mathbf{E}\left[\sum_{v' \in \mathrm{VD}(R_v) b'_{v'} \geq n/s} 1\right] = \sum_{\substack{\mathbf{u} \in B_v^3 \\ b'_{\mathbf{u}} \geq n/s}} \Pr[v'_{\mathbf{u}} \in \mathrm{VD}(R_v)],$$

where $b'_{\mathbf{u}}$ is the conflict size of $v'_{\mathbf{u}}$ with respect to B_v. Using Lemma 3.2 with $X = B_v$, $Y = \emptyset$, and $p = (\alpha t_v \log t_v)/b_v = \alpha(s/n)\log t_v$, this is $O(t_v^{-\alpha/2} t_v \log t_v) \leq 1/4$, for α large enough (remember that $t_v \geq 2$). □

By Lemma 3.6 and Markov's inequality, the probability that all vertices from $\mathrm{VD}(R_v)$ have at most n/s points from B_v in their conflict circles is at least $3/4$. If so, we call v *good*. Scanning through S, we can identify the good vertices in time $O(n \log s)$ and space $O(s)$. Let s' be the size of $\mathrm{VD}(R \cup K)$. If we have less than $s'/2$ good vertices, we repeat the process. Since the expected number of good vertices is $3s'/4$, the probability that there are at least $s'/2$ good vertices is at least $1/2$ by Markov's inequality. Thus, in expectation, we need to perform the sampling twice. For the remaining vertices, we repeat the process, but now we take two samples per vertex, decreasing the failure probability to $1/4$. We repeat the process, taking in each round the maximum number of samples that fit into the work space. In general, if we have s'/a_i active vertices in round i, we can take a_i samples per vertex, resulting in a failure probability of 2^{-a_i}. Thus, the expected number of active vertices in round $i+1$ is $s'/a_{i+1} = s'/(a_i 2^{a_i})$. After $O(\log^* s)$ rounds, all vertices are good. To summarize:

Lemma 3.7. *In total expected time $O(n \log s \log^* s)$ and space $O(s)$, we can find sets $R \subseteq S$ and $R_v \subset B_v$ for each vertex $v \in \mathrm{VD}(R' \cup K)$ such that (i) $|R| = s$: (ii) $\sum_{v \in \mathrm{VD}(R \cup K)} |R_v| = O(s)$; and (iii) for every R_v, all vertices of $\mathrm{VD}(R_v)$ have at most n/s points from B_v in their conflict circle.*

We set $R_2 = R \cup \bigcup_{v \in \mathrm{VD}(R \cup K)} R_v$. By Lemma 3.7, $|R_2| = O(s)$. We compute $\mathrm{VD}(R_2 \cup K)$ and triangulate its bounded cells. For a triangle Δ of the triangulation, let $r \in R_2 \cup K$ be the site whose cell contains Δ, and v_1, v_2, v_3 the vertices of Δ. We set $B_\Delta = \{r\} \cup \bigcup_{i=1}^3 B_{v_i}$. Using the next lemma, we show that $|B_\Delta| = O(n/s)$. The proof is in the full version.

Lemma 3.8. *Let $S \subset \mathbb{R}^2$ and $\Delta = \{v_1, v_2, v_3\}$ a triangle in the triangulation of $\mathrm{VD}(S)$. Let $x \in \Delta$. Then any circle C with center x that contains no points from S is covered by the conflict circles of v_1, v_2 and v_3.*

Lemma 3.9. *Any triangle Δ in the triangulation of $\mathrm{VD}(R_2 \cup K)$ has $|B_\Delta| = O(n/s)$.*

Proof. Let v be a vertex of Δ. We show that $b_v = O(n/s)$. Let $\Delta_R = \{v_1, v_2, v_3\}$ be the triangle in the triangulation of $\mathrm{VD}(R)$ that contains v. By Lemma 3.8, we have $B_v \subseteq \bigcup_{i=1}^{3} B_{v_i}$. We consider the intersections $B_v \cap B_{v_i}$, for $i = 1, 2, 3$. If $t_{v_i} < 2$, then $b_{v_i} = O(n/s)$ and $|B_v \cap B_{v_i}| = O(n/s)$. Otherwise, we have sampled a set R_{v_i} for v_i. Let $\Delta_i = \{w_1, w_2, w_3\}$ be the triangle in the triangulation of $\mathrm{VD}(R_{v_i})$ that contains v. Again, by Lemma 3.8, we have $B_v \subseteq \bigcup_{j=1}^{3} B_{w_j}$ and thus also $B_v \cap B_{v_i} \subseteq \bigcup_{j=1}^{3} B_{w_j} \cap B_{v_i}$. However, by construction of R_{v_i}, $|B_{w_j} \cap B_{v_i}|$ is at most n/s for $j = 1, 2, 3$. Hence, $|B_v \cap B_{v_i}| = O(n/s)$ and $b_v = O(n/s)$. □

The following lemma enables us to compute the Voronoi diagram of $R_2 \cup K$ locally for each triangle Δ in the triangulation of $\mathrm{VD}(R_2 \cup K)$ by only considering sites in B_Δ. It is a direct consequence of Lemma 3.8.

Lemma 3.10. *For every triangle Δ in the triangulation of $\mathrm{VD}(R_2 \cup K)$, we have $\mathrm{VD}(S \cup K) \cap \Delta = \mathrm{VD}(B_\Delta) \cap \Delta$.*

Theorem 3.11. *Let S be a planar n-point set. In expected time $O((n^2/s) \log s + n \log s \log^* s)$ and space $O(s)$, we can compute all Voronoi vertices of S.*

Proof. We compute a set R_2 as above. This takes $O(n \log s \log^* s)$ time and space $O(s)$. We triangulate the bounded cells of $\mathrm{VD}(R_2 \cup K)$ and compute a point location structure for the result. Since there are $O(s)$ triangles, we can store the resulting triangulation in the workspace. Now, the goal is to compute simultaneously for all triangles Δ the Voronoi diagram $\mathrm{VD}(B_\Delta)$ and to output all Voronoi vertices that lie in Δ and are defined by points from S. By Lemma 3.10, this gives all Voronoi vertices of $\mathrm{VD}(S)$.

Given a planar m-point set X, the algorithm by Asano et al. finds all vertices of $\mathrm{VD}(X)$ in $O(m)$ scans over the input, with constant workspace [4]. We can perform a simultaneous scan for all sets B_Δ by determining for each point in S all sets B_Δ that contain it. This takes total time $O(n \log s)$, since we need one point location for each $p \in S$ and since the total size of the B_Δ's is $O(n)$. We need $O(\max_\Delta |B_\Delta|) = O(n/s)$ such scans, so the second part of the algorithm needs $O((n^2/s) \log s)$ time. □

As mentioned in the introduction, Theorem 3.11 also lets us report all edges of the Delaunay triangulation of S in the same time bound: by duality, the three sites that define a vertex of $\mathrm{VD}(S)$ also define a triangle for the Delaunay triangulation. Thus, whenever we discover a vertex of $\mathrm{VD}(S)$, we can instead output the corresponding Delaunay edges, while using a consistent tie-breaking rule to make sure that every edge is reported only once.

Acknowledgments. This work began while W. Mulzer, P. Seiferth, and Y. Stein visited the Tokuyama Laboratory at Tohoku University. We would like to thank Takeshi Tokuyama and all members of the lab for their hospitality and for creating a conducive and stimulating research environment.

References

1. T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz. Memory-constrained algorithms for simple polygons. *Comput. Geom.*, 46(8):959–969, 2013.
2. T. Asano, A. Elmasry, and J. Katajainen. Priority queues and sorting for read-only data. In *TAMC*, pages 32–41, 2013.
3. T. Asano and D. Kirkpatrick. Time-space tradeoffs for all-nearest-larger-neighbors problems. In *Proc. 13th Int. Conf. Algorithms and Data Structures (WADS)*, pages 61–72, 2013.
4. T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *J. of Comput. Geom.*, 2(1):46–68, 2011.
5. L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, pages 1–33, 2014.
6. L. Barba, M. Korman, S. Langerman, and R. I. Silveira. Computing the visibility polygon using few variables. *Comput. Geom.*, 47(9):918–926, 2013.
7. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational geometry*. Springer-Verlag, third edition, 2008. Algorithms and applications.
8. A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11:287–297, 1982.
9. 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. (SoCG)*, pages 239–246, 2004.
10. T. M. Chan and E. Y. Chen. Multi-pass geometric algorithms. *Discrete Comput. Geom.*, 37(1):79–102, 2007.
11. B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.
12. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
13. O. Darwish and A. Elmasry. Optimal time-space tradeoff for the 2D convex-hull problem. In *Proc. 22nd Annu. European Sympos. Algorithms (ESA)*, pages 284–295, 2014.
14. A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3:153–174, 1984.
15. S. Har-Peled. Shortest path in a polygon using sublinear space. To appear in SoCG'15.
16. R. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2(1):18–21, 1973.
17. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
18. J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoret. Comput. Sci.*, 12:315–323, 1980.
19. J. I. Munro and V. Raman. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci.*, 165(2):311–323, 1996.
20. J. Pagter and T. Rauhe. Optimal time-space trade-offs for sorting. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 264–268, 1998.
21. I. Pohl. A minimum storage algorithm for computing the median. Technical Report RC2701, IBM, 1969.
22. J. E. Savage. *Models of computation—exploring the power of computing*. Addison-Wesley, 1998.