Delaunay Triangulations in $O(\mathbf{sort}(n))$ Time and More

Kevin Buchin
Department of Mathematics and Computer Science
TU Eindhoven, The Netherlands
kbuchin@win.tue.nl

Wolfgang Mulzer
Department of Computer Science
Princeton University, USA
wmulzer@cs.princeton.edu

Abstract— We present several results about Delaunay triangulations (DTs) and convex hulls in transdichotomous and hereditary settings: (i) the DT of a planar point set can be computed in expected time $O(\text{sort}(n))$ on a word RAM, where $\text{sort}(n)$ is the time to sort n numbers. We assume that the word RAM supports the *shuffle*-operation in constant time; (ii) if we know the ordering of a planar point set in x- and in y-direction, its DT can be found by a randomized algebraic computation tree of expected *linear* depth; (iii) given a universe U of points in the plane, we construct a data structure D for *Delaunay queries*: for any $P \subseteq U$, D can find the DT of P in time $O(|P| \log\log |U|)$; (iv) given a universe U of points in 3-space in general convex position, there is a data structure D for *convex hull queries*: for any $P \subseteq U$, D can find the convex hull of P in time $O(|P|(\log\log |U|)^2)$; (v) given a convex polytope in 3-space with n vertices which are colored with $\chi > 2$ colors, we can split it into the convex hulls of the individual color classes in time $O(n(\log\log n)^2)$. The results (i)–(iii) generalize to higher dimensions. We need a wide range of techniques. Most prominently, we describe a reduction from DTs to nearest-neighbor graphs that relies on a new variant of randomized incremental constructions using *dependent* sampling.

1. Introduction

Everyone knows that it takes $\Omega(n \log n)$ time to sort n numbers—and yet this lower bound can often be beaten. Under the right assumptions, radix sort and bucket sort run in linear time [25]. Using van Emde Boas (vEB) trees [43], we can sort n elements from a universe U in $O(n \log\log |U|)$ time on a pointer machine. In a *transdichotomous* model, we can surpass the sorting lower bound with *fusion trees*, achieving $O(n\sqrt{\log n})$ time. Fusion trees were introduced in 1990 by Fredman and Willard [30] and triggered off a development that culminated in the $O(n\sqrt{\log\log n})$ time sorting algorithm by Han and Thorup [31]. For small ($O(\log n)$) and large ($\Omega(\log^{2+\varepsilon} n)$) word sizes, we can even sort in linear time (via radix sort [25], resp. signature sort [6]).

In computational geometry, there have been many results that use vEB trees or similar structures to surpass traditional lower bounds (eg, [5], [21], [27], [32], [33], [40]). However, these results assume that the input is rectilinear or can be efficiently approximated by a rectilinear structure, like, for example, a quadtree. In this sense, the above results are all *orthogonal*. Similarly, Willard [45] applied fusion trees to achieve better bounds for orthogonal range searching, axis-parallel rectangle intersection, and others. Again, his results are all orthogonal, and he asked whether improved bounds can be attained for Voronoi diagrams. The breakthrough came in 2006, when Chan and Pătraşcu [16] discovered transdichotomous algorithms for point location in non-orthogonal planar subdivisions. This led to better bounds for many classical computational geometry problems. In a follow-up paper [17], they considered off-line planar point location and thereby improved the running time for Delaunay triangulations, three-dimensional convex hulls, and other problems. The running time is a rather unusual $n2^{O(\sqrt{\log\log n})}$, which raises the question whether the result is optimal. More generally, Chan and Pătraşcu asked if the approach via point location is inherent, or if there are more direct algorithms for convex hulls or Delaunay triangulations. Finally, they also briefly discussed [16, Section 3] the merits of various approaches to transdichotomous algorithms and why fusion trees seemed most feasible for planar point location. In particular, this leaves the question open whether the vEB approach can provide any benefits for computational geometry, apart from the orthogonal results mentioned above.

Let us compare the respective properties of the vEB and the fusion method. The latter is more recent and makes stronger use of the transdichotomous model. The main idea is to "fuse" (parts of) several data items into one word and then use constant time bit-operations for parallel processing. Unfortunately, since results based on fusion trees need all the transdichotomous power, they usually do not generalize to other models of computation. The vEB method, on the other hand, is the older one and is essentially based on hashing: the data is organized as a tree of high degree, and the higher-order bits of a data item are used to locate the appropriate child at each step. Thus, they apply in any model in which the hashing step can be performed in constant time, for example a pointer machine [36]. This becomes particularly useful for *hereditary* results [20], where we would like to preprocess a large universe U in order to quickly answer queries about subsets of U. In this setting, vEB trees show that it suffices to sort a big set once in order to sort any large enough subset of U faster than in $\Theta(n \log n)$ time.

Our results. We begin with a randomized reduction from Delaunay triangulations (DTs) to nearest-neighbor graphs (NNGs). Our method uses a new variant of the classic

randomized incremental construction (RIC) paradigm [4], [24], [39] that relies on dependent sampling for faster conflict location. If NNGs can be computed in linear time, the running time of our reduction is proportional to the structural change of a standard RIC, which is always linear for planar point sets and also in many other cases, eg, point sets suitably sampled from a $(d-1)$-dimensional polyhedron in $\mathbb{R}^d$ [3], [8].[1] The algorithm is relatively simple and works in any dimension, but the analysis turns out to be rather subtle. It is a well-known fact that given a quadtree for a point set, its nearest-neighbor graph can be computed in linear time [13], [15], [22]. This leads to our main discovery: *Given a quadtree for a point set $P \subseteq \mathbb{R}^d$, we can compute the Delaunay triangulation of P, $\mathrm{DT}(P)$, in time proportional to the expected structural change of a RIC.* This may be surprising, since even though DTs appear to be inherently non-orthogonal, we actually need only the information encoded in quadtrees, a highly orthogonal structure.

Our reduction has many consequences. First, we answer Willard's seventeen-year-old open question by showing that planar DTs, and hence planar Voronoi diagrams and related structures like Euclidean minimum spanning trees, can be computed in time $O(\mathtt{sort}(n))$ on a word RAM. Our algorithm requires one non-standard, but AC^0, operation, the *shuffle*. However, we believe that it should be straightforward to adapt existing integer sorting algorithms so that our result also holds on a more standard word RAM. In Appendix A, we demonstrate this for the (comparatively simple) $O(n \log\log n)$ sorting algorithm by Andersson *et al* [6]. Since our reduction works in a traditional model, we also get pointer machine algorithms for hereditary DTs: we can preprocess a point set U such that for any subset $P \subseteq U$, it takes $O(|P| \log\log |U| + C(P))$ time to find $\mathrm{DT}(P)$. Here, $C(P)$ denotes the expected structural change of a RIC on P. Since a planar quadtree can be computed by an algebraic computation tree (ACT) of linear depth once the points are sorted according to the x- and y-direction, we find that after presorting in *two* orthogonal directions, a planar DT can be computed by an ACT of expected linear depth. This should be compared with the fact that there is an $\Omega(n \log n)$ lower bound when the points are sorted in *one* direction [28], and also for convex hulls in $\mathbb{R}^3$ when the points are sorted in any constant number of directions [42]. This problem has appeared in the literature for at least twenty years [1], [21], [28], and our result seems to mark the first non-trivial progress on this question. However, we do not know if a quadtree for presorted points can be constructed in linear time, since the algorithms we know still need an $\Omega(n \log n)$ overhead for data handling.[2]

In the second part, we extend the result about hereditary DTs to 3-polytopes and describe a vEB-like data structure for this problem: preprocess a point set $U \subseteq \mathbb{R}^3$ in general convex position such that the convex hull of any $P \subseteq U$ can be found in time $O(|P|(\log\log |U|)^2)$. These queries are called *convex hull queries*. We use a relatively recent technique [20], [23], [44] which we call *scaffolding*: in order to find many related structures quickly, we first compute a "typical" instance—the *scaffold* S—in a preprocessing phase. To answer a query, we insert the input points into S and use a fast *hereditary* algorithm [20] to remove the scaffold. We also need a carefully balanced recursion and a bootstrapping method similar to the one by Chan and Pătraşcu [17]. This also improves a recent algorithm for splitting a 3-polytope whose vertices are colored with $\chi \geq 2$ colors into its monochromatic parts [20, Theorem 4.1]. All our algorithms are randomized, and it is an interesting open problem to derandomize them. In particular, it would be very interesting to find a deterministic algorithm for splitting DTs [18] or, more generally, convex polytopes [20].

Computational models. We use two different models of computation, a *word RAM* and a *pointer machine*. The former represents data as a sequence of w-bit words, where $w = \Omega(\log n)$. Data can be accessed randomly, and standard operations (ie, Boolean operations, addition, or multiplication) take constant time. We need one nonstandard operation: given a point $p \in \mathbb{R}^d$ with w-bit coordinates $p_{1w} \dots p_{12} p_{11}$, $p_{2w} \dots p_{21}$, …, $p_{dw} \dots p_{d1}$, the result of $\mathtt{shuffle}(p)$ is the dw-bit word $p_{1w} p_{2w} \dots p_{dw} \dots p_{12} \dots p_{d2} p_{11} \dots p_{d1}$. Clearly, $\mathtt{shuffle}$ is in AC^0, and we assume that it takes constant time on our RAM. In Appendix A, we indicate how this assumption can be dropped.

On a *pointer machine*, the data structure is modeled as a directed graph G with bounded out-degree. Each node in G represents a *record*, with a bounded number of pointers to other records and a bounded number of (real or integer) data items. For each point in the universe U there is a record storing its coordinates, and the input sets are provided as a linked list of records, each pointing to the record for the corresponding input. The output is provided as a DCEL [26, Chapter 2.2]. The algorithm can access data only by following pointers from the inputs (and a bounded number of global entry records); random access is not possible. The data can be manipulated through the usual real RAM operations, such as addition, multiplication, or square-root. However, we assume that the floor function is not supported, to prevent our computational model from becoming too powerful [41].

[1] The bound in the references is only proved for the complexity of the final DT, but we believe that it can be extended to the structural change of a RIC.

[2] It would be interesting to see if there is a connection to the notorious SORTING $X + Y$ problem [29], which seems to exhibit a similar behavior.

2. From NNGs to DTs

We now describe our reduction from nearest-neighbor graphs (NNGs) to Delaunay triangulations (DTs). This is done by a randomized algorithm which we call `BrioDC`, for Biased Random Insertion Order with Dependent Choices:

Algorithm `BrioDC`(P)
1) If $|P| = O(1)$, compute $\mathrm{DT}(P)$ directly and return.
2) Compute $\mathrm{NN}(P)$.
3) Let $S \subseteq P$ be a random sample such that (i) S meets every connected component of $\mathrm{NN}(P)$ and (ii) $\Pr[p \in S] = 1/2$, for all $p \in P$.
4) Call `BrioDC`(S) to compute $\mathrm{DT}(S)$.
5) Compute $\mathrm{DT}(P)$ by inserting the points in $P \setminus S$ into $\mathrm{DT}(S)$, using $\mathrm{NN}(P)$ as a guide.

Algorithm 1: The reduction from DTs to NNGs.

To find S in Step 3, we define a partial matching $\mathcal{M}(P)$ on P by pairing up two arbitrary points in each component of $\mathrm{NN}(P)$, the nearest-neighbor graph of P. Then, S is obtained by picking one random point from each pair in $\mathcal{M}(P)$ and sampling the points in $P \setminus \mathcal{M}(P)$ independently with probability $1/2$ (although they could also be paired up). In Step 5, we successively insert the points from $P \setminus S$ as follows: pick a point $p \in P \setminus S$ that has not been inserted yet and is adjacent in $\mathrm{NN}(P)$ to a point q in the current DT. Such a point always exists by the definition of S. Walk along the edge qp to locate p in the current DT, and insert it. Repeat until all of P has been processed.

Theorem 2.1. *Suppose the nearest-neighbor graph of an m-point set can be found in $f(m)$ time, where $f(m)/m$ is increasing. Let $P \subseteq \mathbb{R}^d$ be an n-point set. The expected running time of `BrioDC` is $O(C(P)+f(n))$, where $C(P)$ is the expected structural change of a RIC on P. The constant in the O-notation depends exponentially on d.*

Let $P = S_0 \supseteq \cdots \supseteq S_\ell$ be the sequence of samples. Fix a set $\mathbf{u}$ of $d+1$ distinct points in P. Let Δ be the simplex spanned by $\mathbf{u}$, and let $L_{\mathbf{u}} \subseteq P$ denote the points inside Δ's circumsphere. We call $\mathbf{u}$ the *trigger* set and $L_{\mathbf{u}}$ the *stopper* set for Δ. Consider the event A_α that Δ occurs during the construction of $\mathrm{DT}(S_\alpha)$ from $\mathrm{DT}(S_{\alpha+1})$, for some α. Clearly, A_α can only happen if $\mathbf{u} \subseteq S_\alpha$ and $L_{\mathbf{u}} \cap S_{\alpha+1} = \emptyset$. To prove Theorem 2.1, we bound $\Pr[A_\alpha]$.

Lemma 2.2. *We have*

$$\Pr[A_\alpha] \leq e^{2d+2}\, 2^{-(d+1)\alpha} \left(1 - 2^{-\alpha-1}\right)^{|L_{\mathbf{u}}|}.$$

We visualize the sampling process as follows [38, Chapter 1.4]: imagine a particle that moves at discrete time steps on the nonnegative x-axis and always occupies integer points. The particle starts at position $|L_{\mathbf{u}}|$, and after β steps, it is at position $|S_\beta \cap L_{\mathbf{u}}|$, the number of stoppers in the current sample. The goal is to bound the probability of reaching 0 in $\alpha+1$ steps, retaining all the triggers. However, the random choices in a step not only depend on the current position, but also on the matching $\mathcal{M}(S)$. Even worse, the probability distribution in the current position may depend on the previous positions of the particle. We avoid these issues through appropriate conditioning and show that the random walk essentially behaves like a Markov process that in each round eliminates $d+1$ stoppers and samples the remaining stoppers independently. The elimination is due to trigger-stopper pairs in $\mathcal{M}(S)$, since we want all triggers to survive. The remaining stoppers are not necessarily independent, but dependencies can only help, because in each stopper-stopper pair one stopper is guaranteed to survive. Eliminating $d+1$ stoppers in the ith step has a similar effect as starting with about $(d+1)2^i$ fewer stoppers: though a given trigger can be matched with only one stopper per round, these pairings can vary for different instances of the walk, and since a given stopper survives a round with probability roughly $1/2$, the "amount" of stoppers eliminated by one trigger in all instances roughly doubles per round.

Proof of Lemma 2.2: For a subset $S \subseteq P$, define the *matching profile* for S as the triple (a,b,c) that counts the number of trigger-stopper, stopper-stopper, and trigger-trigger pairs in $\mathcal{M}(S)$. We consider $p_{s,k} = \max_{\mathcal{P}_k} \Pr[A_\alpha \mid X_{s,k}, \mathcal{P}_k](*)$, where $X_{s,k} = \{\mathbf{u} \subseteq S_{\alpha-k}\} \cap \{|L_{\mathbf{u}} \cap S_{\alpha-k}| = s\}$ is the event that the sample $S_{\alpha-k}$ contains all triggers and exactly s stoppers. The maximum in (*) is taken over all possible sequences $\mathcal{P}_k = \mathbf{m}_0, \ldots, \mathbf{m}_{\alpha-k-1}, Y_0, \ldots, Y_{\alpha-k-1}$ of matching profiles $\mathbf{m}_i$ for S_i and events Y_i of the form $X_{t_i,\alpha-i}$ for some t_i. Since $\Pr[A_\alpha] = p_{|L_{\mathbf{u}}|,\alpha}$, it suffices to upperbound $p_{s,k}$. We describe a recursion for $p_{s,k}$. For that, let $T_k = \{\mathbf{u} \subseteq S_{\alpha-k}\}$ be the event that $S_{\alpha-k}$ contains all the triggers, and let $U_{k,i} = \{|L_{\mathbf{u}} \cap S_{\alpha-k}| = i\}$ denote the event that $S_{\alpha-k}$ contains exactly i stoppers.

Proposition 2.3. *We have*

$$p_{s,k} \leq \max_{\boldsymbol{m}} \Pr[T_{k-1} \mid X_{s,k}, \boldsymbol{m}] \cdot \sum_{i=0}^{s} p_{i,k-1} \Pr[U_{k-1,i} \mid T_{k-1}, X_{s,k}, \boldsymbol{m}],$$

where the maximum is over all possible matching profiles $\boldsymbol{m} = (a,b,c)$ for $S_{\alpha-k}$.

Proof: Fix a sequence $\mathcal{P}_k$ as in (*). Then, by distinguishing how many stoppers are present in $S_{\alpha-k+1}$,

$$\Pr[A_\alpha \mid X_{s,k}, \mathcal{P}_k] = \sum_{i=0}^{s} \Pr[X_{i,k-1} \mid X_{s,k}, \mathcal{P}_k] \Pr[A_\alpha \mid X_{i,k-1}, X_{s,k}, \mathcal{P}_k].$$

Now if we condition on a matching profile $\mathbf{m}$ for $S_{\alpha-k}$, we

get

$$\Pr[X_{i,k-1} \mid X_{s,k}, \mathcal{P}_k, \mathbf{m}] = \Pr[T_{k-1} \mid X_{s,k}, \mathbf{m}] \Pr[U_{k-1,i} \mid T_{k-1}, X_{s,k}, \mathbf{m}],$$

since the distribution of triggers and stoppers in $S_{\alpha-k+1}$ becomes independent of $\mathcal{P}_k$ once we know the matching profile and the number of triggers and stoppers in $S_{\alpha-k}$. Furthermore,

$$\Pr[A_\alpha \mid X_{i,k-1}, \mathbf{m}, X_{s,k}, \mathcal{P}_k] \leq \max_{\mathcal{P}_{k+1}} \Pr[A_\alpha \mid X_{i,k-1}, \mathcal{P}_{k+1}] = p_{i,k-1}.$$

The claim follows by taking the maximum over $\mathbf{m}$. ∎

We use Proposition 2.3 to bound $p_{s,k}$: if $\mathbf{m} = (a, b, c)$ pairs up two triggers, we get $\mathbf{u} \not\subseteq S_{\alpha-k+1}$ and $\Pr[T_{k-1} \mid X_{s,k}, \mathbf{m}] = 0$. Hence we can assume $c = 0$ and therefore $\Pr[T_{k-1} \mid X_{s,k}, \mathbf{m}] = 1/2^{d+1}$, since all triggers are sampled independently. Furthermore, none of the a stoppers paired with a trigger and half of the $2b$ stoppers paired with a stopper end up in $S_{\alpha-k+1}$, while the remaining $t_\mathbf{m} = s-a-2b$ stoppers are sampled independently. Thus, Proposition 2.3 gives

$$p_{s,k} \leq \max_{\mathbf{m}} \sum_{c=0}^{s-a-b} \sum_{i=b} \frac{p_{i,k-1}}{2^{d+1}} \Pr\left[B_{1/2}^{t_\mathbf{m}} = i - b\right], \qquad (1)$$

where $B_{1/2}^{t_\mathbf{m}}$ denotes a binomial distribution with $t_\mathbf{m}$ trials and success probability $1/2$.

Proposition 2.4. *We have*

$$p_{s,k} \leq 2^{-(d+1)k} \left(1 - 2^{-k-1}\right)^s \prod_{j=1}^{k} \left(1 - 2^{-j}\right)^{-d-1}.$$

Proof: The proof is by induction on k. For $k = 0$, we have $p_{s,0} \leq (1 - 1/2)^s$, since we require that none of the s stoppers in S_α be present in $S_{\alpha+1}$, and this can only happen if they are sampled independently of each other. Furthermore, by (1),

$$\begin{aligned} p_{s,k+1} &\leq \max_{\mathbf{m}} \sum_{c=0}^{s-a-b} \sum_{i=b} \frac{p_{i,k}}{2^{d+1}} \Pr\left[B_{1/2}^{t_\mathbf{m}} = i - b\right] \\ &= \max_{\mathbf{m}} \frac{1}{2^{d+1+t_\mathbf{m}}} \sum_{i=0}^{t_\mathbf{m}} \binom{t_\mathbf{m}}{i} p_{i+b,k}. \qquad (2) \end{aligned}$$

Using the inductive hypothesis and the binomial theorem, we bound the sum as

$$\begin{aligned} &\sum_{i=0}^{t_\mathbf{m}} \binom{t_\mathbf{m}}{i} p_{i+b,k} \\ &\leq \frac{\sum_{i=0}^{t_\mathbf{m}} \binom{t_\mathbf{m}}{i} \left(1 - 2^{-k-1}\right)^{i+b}}{2^{(d+1)k}} \prod_{j=1}^{k} \left(1 - 2^{-j}\right)^{-d-1} \\ &= \frac{\left(2 - 2^{-k-1}\right)^{t_\mathbf{m}}}{2^{(d+1)k}} \left(1 - 2^{-k-1}\right)^b \prod_{j=1}^{k} \left(1 - 2^{-j}\right)^{-d-1} \\ &= \frac{\left(1 - 2^{-k-2}\right)^{t_\mathbf{m}}}{2^{(d+1)k-t_\mathbf{m}}} \left(1 - 2^{-k-1}\right)^b \prod_{j=1}^{k} \left(1 - 2^{-j}\right)^{-d-1}. \end{aligned}$$

Now, since $t_\mathbf{m} = s - a - 2b \geq s - d - 1 - 2b$ and since

$$\frac{\left(1 - 2^{-k-1}\right)^b}{\left(1 - 2^{-k-2}\right)^{2b}} = \left(\frac{1 - 2^{-k-1}}{1 - 2^{-k-1} + 2^{-2k-4}}\right)^b \leq 1,$$

it follows that

$$\sum_{i=0}^{t_\mathbf{m}} \binom{t_\mathbf{m}}{i} p_{i+b,k} \leq \frac{\left(1 - 2^{-k-2}\right)^s}{2^{(d+1)k-t_\mathbf{m}}} \prod_{j=1}^{k+1} \left(1 - 2^{-j}\right)^{-d-1},$$

and hence (2) gives

$$p_{s,k+1} \leq \frac{\left(1 - 2^{-k-2}\right)^s}{2^{(d+1)(k+1)}} \prod_{j=1}^{k+1} \left(1 - 2^{-j}\right)^{-d-1},$$

which finishes the induction. ∎

Now, since $1 - x \geq \exp(x/(x-1))$ for $x < 1$ we have $\prod_{j=1}^{k}(1 - 2^{-j})^{-d-1} \leq e^{2(d+1)\sum_{j=1}^{\infty} 2^{-j}} = e^{2(d+1)}$, so $\Pr[A_\alpha] \leq p_{|L_\mathbf{u}|,\alpha} \leq e^{2d+2} 2^{-(d+1)\alpha} (1 - 2^{-\alpha-1})^{|L_\mathbf{u}|}$, which proves Lemma 2.2. ∎

Proof of Theorem 2.1: Since $f(m)/m$ increases, the expected cost to compute the NNGs for all the samples is $O(n)$. Furthermore, the cost of tracing an edge pq of $\mathrm{NN}(P)$, where p is in the current DT and q will be inserted next consists of (a) the cost of finding the starting simplex at p and (b) the cost of walking through the DT. Part (a) can be bounded by the degree of p in the current DT. In total, any simplex appears at most as often as the total degree of its vertices in $\mathrm{NN}(P)$, which is constant [37, Corollary 3.2.3]. Hence, (a) is proportional to the structural change. The same holds for (b), since every traversed simplex will be destroyed when the next point is inserted.

It is now sufficient to show that the probability that the simplex spanned by $\mathbf{u} \subseteq P$ occurs in `BrioDC` is asymptotically upperbounded by the corresponding probability in a RIC. In the case of `BrioDC`, this probability is bounded by $\sum_{\alpha=0}^{\infty} \Pr[A_\alpha]$. Let p denote the corresponding probability in a RIC and let $p_\alpha = 2^{-(d+1)\alpha}(1-2^{-\alpha-1})^{|L_\mathbf{u}|}(1-2^{-d-1})$. We have $\Pr[A_\alpha] \leq \exp(2d+2)(1 - 2^{-d-1})^{-1} p_\alpha$. Now, from an analysis of RIC con BRIO [12, Lemma 3.8] we have, $\sum_{\alpha=0}^{\infty} p_\alpha \leq 2^{d+1} p$, thus, $\sum_{\alpha=0}^{\infty} \Pr[A_\alpha] \leq 2^{d+1} \exp(2d + 2)(1 - 2^{-d-1})^{-1} p$, as desired. ∎

Remark. The reduction also shows that it takes $\Omega(n \log n)$ time to compute NNGs and well-separated pair decompositions, even if the input is sorted along one direction [28].

Remark. The dependent sampling has more advantages than just allowing for fast point location. For instance, if P samples a region, eg, a surface [8], in the sense that for any point in the region there is a point in P at distance at most ε, then similar guarantees with increasing ε still hold for $S = S_1, S_2, \ldots$. Furthermore, Lemma 2.2 directly generalizes to the more general setting of *configuration spaces* [39] by replacing $d+1$ by the degree bound, ie, the maximum number of triggers. Thus, our dependent sampling scheme can be used in the incremental construction of a wide range of structures, and may be useful in further applications.

Remark. Finally, we note that if P is planar, the proof of Theorem 2.1 can be simplified considerably:

A simple proof of Theorem 2.1 for $d = 2$: As we argued at the beginning of the proof of Theorem 2.1, it suffices to bound the structural change. For this, we compare it to the structural change of the last round of a RIC con BRIO [4]. Let p_s be the probability that a given triangle with s conflicts appears in the last round of such a construction. We have $p_s = c/2^s$ for a suitable constant c. The probability p'_s that this triangle appears in `BrioDC` while constructing $\mathrm{DT}(P)$ from $\mathrm{DT}(S)$ is also bounded by $1/2^s$: either the stoppers of the triangle are sampled independently of each other (then we directly get this bound), or not (then S includes a stopper and the simplex cannot occur). Thus, the expected structural change is asymptotically as for RIC con BRIO and therefore linear [4], [12]. Now, the expected size of S is $|P|/2$, and we can apply the argument above to the construction of $DT(S)$, and so on. Overall this yields the desired running time. Note that this argument does not apply in higher dimensions, because the sample S is biased and so without further argument we cannot exclude the possibility that the complexity of $\mathrm{DT}(S)$ is large, even if $C(P)$ is small. ■

3. Delaunay triangulations

Let $P \subseteq \mathbb{R}^d$ be an n-point set whose coordinates are w-bit words. The *shuffle*-order of P is obtained by taking $\texttt{shuffle}(p)$ for every $p \in P$, as described above, and sorting the resulting numbers in the usual order. The shuffle order is intimately related to quadtrees [11], [15].

Lemma 3.1. *Suppose our computational model is a word RAM. Let $P \subseteq \mathbb{R}^d$ be given in shuffle-order. Then, a compressed quadtree for P can be computed in $O(|P|)$ time.*

Proof: Our argument mostly follows Chan's presentation [15, Step 2]. We define a hierarchy H of quadtree-boxes, by taking the hypercube $[0, 2^w - 1]$ as the root box and by letting the children of a box b be the hypercubes that divide b into 2^d equal axis-parallel parts. For two points p, q, let $\texttt{box}(p, q)$ be the smallest quadtree box that contains p and q, and let $|\texttt{box}(p, q)|$ be the side-length of this box. Both can be found by examining the most significant bits in which the coordinates of p and q differ. A *compressed quadtree* for a point set P is the subtree of H induced by the leaves in H that correspond to P. The crucial observation that connects compressed quadtrees with the shuffle order is that if the children of each node in H are ordered lexicographically, then the leaves of H are sorted according to the shuffle order. The quadtree is constructed by `BuildQuadTree` (Algorithm 2). Correctness and running time follow just as in Chan's paper, if our model supports the `msb` (most significant bit) operation in constant time, where `msb` is the index of the first nonzero bit in a word.

Algorithm `BuildQuadTree`

1) q_0.box $= \mathbb{R}^d$, q_0.children $= (p_1)$, $k = 0$
2) for $i = 2, \ldots, n$
   a) while $|\texttt{box}(p_{i-1}, p_i)| > |q_k.\text{box}|$ do $k = k - 1$
   b) if $|q_k.\text{box}| = |\texttt{box}(p_{i-1}, p_i)|$, let p_i be the next child of q_k; otherwise, create q_{k+1} with $q_{k+1}.\text{box} = \texttt{box}(p_{i-1}, p_i)$, and move the last child of q_k to the first child of q_{k+1}, make p_i the second child of q_{k+1}, and q_{k+1} the last child of q_k. Set $k = k + 1$.

Algorithm 2: Building a compressed quadtree.

With some more effort, we can avoid the `msb`-operation. First, note that box-sizes can be compared without `msb`, by using (4) as in Appendix A. Then `BuildQuadTree` finds the combinatorial structure of the compressed quadtree T for P. Using a post-order traversal of T, we can find for each node b in T a minimum bounding box for the points under b, in linear time. This information suffices to apply Lemma 3.2, as we can see by inspecting the proof of Callahan and Kosaraju [13]. ■

Via well-separated pair decompositions, we can go from quadtrees to NNGs in linear time [13], [15], [22].

Lemma 3.2. *Let $P \subseteq \mathbb{R}^d$. Given a compressed quadtree for P, we can find $\mathrm{NN}(P)$ in $O(|P|)$ time in a traditional model (and also on a word RAM).* □

Combining Theorem 2.1 with Lemmas 3.1 and 3.2, we derive the main result for this section.

Theorem 3.3. *Suppose our computational model is a word RAM with a constant-time shuffle operation. Let $P \subseteq \mathbb{R}^d$ be an n-point set. Then $\mathrm{DT}(P)$ can be computed in expected time $O(\mathit{sort}(n) + C(P))$, where $C(P)$ denotes the expected structural change of a RIC on P and $\mathit{sort}(n)$ denotes the time needed for sorting n numbers.* □

Remark. For planar point sets, $C(P)$ is always linear, and this also often holds in higher dimensions. Furthermore,

in the plane there is another approach to Theorem 3.3, which we sketch here: we sort P in shuffle order and compute a quadtree for P using Lemma 3.1. Then we use the techniques of Bern *et al* [10], [11] to find a point set $P' \supseteq P$ and $\mathrm{DT}(P')$ in $O(n)$ time, where $|P'| = O(n)$. Finally, we extract $\mathrm{DT}(P)$ with a linear-time algorithm for splitting Delaunay triangulations [18], [20].

Our reduction has a curious consequence about presorted point sets, since quadtrees for such point sets can be found by an algebraic computation tree [7, Chapter 14] of linear depth.

Theorem 3.4. *Let $P \subseteq \mathbb{R}^d$ be an n-point set, such that the order of P along each coordinate axis is known. Then $\mathrm{DT}(P)$ can be computed by an algebraic computation tree with expected depth $O(sort(n) + C(P))$, where $C(P)$ denotes the expected structural change of a RIC on P.*

Proof (sketch): Build the quadtree in the standard way [10], [13], but use simultaneous exponential searches from both sides when partitioning the points in each box. In each step of the search, we compare a coordinate of an input point with the average of the corresponding coordinates of two other input points. Then, we see that the number of such comparisons obeys a recursion of the type $T(n) = O(\log(\min(n_1, n_2)) + T(n_1) + T(n_2)$, with $n_1, n_2 \leq n - 1$ and $n_1 + n_2 = n$, which solves to $T(n) = O(n)$. This recursion holds only for nodes in which we are making progress in splitting the point set, but in all other nodes we perform only constantly many comparisons and there are linearly many of them. Nonetheless, we still need $O(n \log n)$ steps to split both the x- and the y-lists while building the tree. ∎

We can also use our reduction to get a vEB-like result for planar Delaunay triangulations.

Theorem 3.5. *Let $U \subseteq \mathbb{R}^d$ be a u-point set. In $O(u \log u)$ time we can preprocess U into a data structure for the following kind of queries: given $P \subseteq U$ with n points, compute $\mathrm{DT}(P)$. The time to answer a query is $O(n \log \log u + C(P))$. The algorithm runs on a traditional pointer machine.*

Proof: Compute a compressed quadtree T for U in time $O(u \log u)$ [10]. We use T in order to find NNGs quickly. Let $S \subseteq U$ be a subset of size m. The *induced subtree* for S, T_S, is the union of all paths from the root of T to a leaf in S. It can be found in time $O(m \log \log u)$.

Claim 3.6. *We can preprocess T into a data structure of size $O(u \log \log u)$ such that for any subset $S \subseteq U$ of m points we can compute the induced subtree T_S in time $O(m \log \log u)$.*

Proof: Build a vEB tree [36], [43] A for U, and preprocess T into a pointer-based data structure for least-common-ancestor (lca) queries [35]. Furthermore, for each node of T compute its depth, and build a vEB priority queue B for the depths in T. These data structures need $O(u \log \log u)$ space. Given S, use the vEB tree A to sort S according to the order of T. Then use the lca-structure to compute T_S as follows: initialize a linked list Q with S in sorted order. Insert the elements in S into vEB tree B, using the depth of the corresponding leaves as the key. Remove from B an element v with maximum depth, use Q to find v's two neighbors, and perform two lca-queries, one on v and its left neighbor, one on v and its right neighbor. Replace v in Q and B by the lower of the two ancestors (or delete v, if the ancestor is already present). Since there are $O(m)$ queries to the vEB trees, and $O(m)$ queries to the lca-structure, the whole process takes time $O(m \log \log u)$, as claimed. ∎

In order to find $\mathrm{NN}(S)$, use Claim 3.6 to compute T_S and then use Lemma 3.2. This takes $O(m \log \log m)$ time, and now the claim follows from Theorem 2.1. ∎

Remark. As is well known, once we have computed the DT, we can find many other important geometric structures in $O(n)$ time, see the paper by Chan and Pătraşcu [16].

4. Scaffold trees

We now extend Theorem 3.5 to three-dimensional convex hulls and describe a data structure that allows us to quickly find the convex hull of subsets of a large point set $U \subseteq \mathbb{R}^3$ in general and convex position (gcp). We call our data structure the *scaffold tree*. Our description starts with a simple structure that handles convex hull queries in time $O(n\sqrt{\log u} \log \log u)$, which we then bootstrap for the final result.

4.1. The basic structure

For a point set $U \subseteq \mathbb{R}^3$, let $\mathrm{conv}\, U$ denote the convex hull of U, and let $E[U], F[U]$ be the edges and facets of $\mathrm{conv}\, U$. Let $S \subseteq U$, and for a facet $f \in F[S]$, let h_f^+ denote the open half-space whose bounding hyperplane is spanned by f and which does not contain S. For a point $p \in U \setminus S$ and a facet $f \in F[S]$, we say that p is *in conflict* with f if $p \in h_f^+$. For $f \in F[S]$, let $B_f \subseteq U$ denote the points in conflict with f, the *conflict set* of f. Similarly, for $p \in U$, we let the conflict set $D_p \subseteq F[S]$ be the facets in $\mathrm{conv}\, S$ in conflict with p. Furthermore, we set $b_f = |B_f|$ and $d_p = |D_p|$, the *conflict sizes* of f and p. We will need a recent result about splitting convex hulls [20, Theorem 2.1].

Theorem 4.1. *Let $U \subseteq \mathbb{R}^3$ be a u-point set in gcp, and let $P \subseteq U$. There exists an algorithm `SplitHull`, that, given $\mathrm{conv}\, U$, computes $\mathrm{conv}\, P$ in expected time $O(u)$.* □

We will also need a sampling lemma for the recursive construction of the tree [20, Lemma 4.2].

Lemma 4.2. *Let $U \subseteq \mathbb{R}^3$ be a u-point set in gcp, and let $\alpha \in \{1, \ldots, u\}$. Given $\mathrm{conv}\, U$, in $O(u)$ time we can compute subsets $S, Q \subseteq U$ and a partition $Q_1, \ldots, Q_\beta$ of Q such that*

1) $|S| = \alpha$, $|Q| = \Omega(u)$, $\max_i |Q_i| = O\left(\frac{u}{\alpha}\log\alpha\right)$.
2) *For $i = 1, \ldots, \beta$, there is a facet $f_i \in F[S]$ such that all points in Q_i are in conflict with f_i.*
3) *Every point in Q conflicts with constantly many facets of* $\operatorname{conv} S$.
4) *The conflict sets for points $p \in Q_i$, $q \in Q_j$, $i \neq j$, are disjoint and no conflict facet of p shares an edge with a conflict facet of q.*

We can compute $\operatorname{conv} S$, $\operatorname{conv} Q_1$, ..., $\operatorname{conv} Q_\beta$, *and* $\operatorname{conv}(U \setminus (Q \cup S))$ *in expected time $O(u)$.* □

Theorem 4.3. *Let $U \subseteq \mathbb{R}^3$ be a u-point set in gcp. In $O(u \log u)$ time, we can construct a data structure of size $O(u\sqrt{\log u})$ such that for any n-point set $P \subseteq U$ we can compute* $\operatorname{conv} P$ *in time $O(n\sqrt{\log u}\log\log u)$. If* $\operatorname{conv} U$ *is given, the preprocessing time is $O(u\sqrt{\log u})$.*

Proof: We describe the preprocessing phase. If needed, we construct $\operatorname{conv} U$ in time $O(u \log u)$. The scaffold tree is computed through a recursive procedure `BuildTree`(U):

Algorithm `BuildTree`(U)
1) If $|U| = O(1)$, store U and return, otherwise, let $U_1 = U$ and $i = 1$
2) While $|U_i| > u/2^{\sqrt{\log u}}$.
   a) Apply Lemma 4.2 to U_i with $\alpha = 2^{\sqrt{\log |U_i|}}$ to obtain subsets $S_i, R_i \subseteq U_i$, as well as a partition $\mathcal{P}_i = \left(R_i^{(j)}\right)_{1 \le j \le \beta}$ of R_i, and the hulls $\operatorname{conv} S_i$, $\operatorname{conv} R_i^{(j)}$ described in the lemma.
   b) Call `BuildTree`$\left(R_i^{(j)}\right)$ for $j = 1, \ldots, \beta$.
   c) Let $U_{i+1} = U_i \setminus (S_i \cup R_i)$. Use `SplitHull` to compute $\operatorname{conv} U_{i+1}$, and increment i.
3) Let $\ell = i$ and call `BuildTree`(U_ℓ).

Algorithm 3: Building the basic scaffold tree.

By Lemma 4.2, the sizes of the sets U_i decrease geometrically, so we get $\ell = O(\sqrt{\log u})$ and

$$\sum_{i=1}^{\ell} |S_i| = O\left(2^{\sqrt{\log u}}\sqrt{\log u}\right). \tag{3}$$

By Lemma 4.2 and Theorem 4.1, the total time for Steps 2a and 2c is $O(u)$. Since the sets for the recursive calls in Steps 2b and 3 have size $O(u\sqrt{\log u}/2^{\sqrt{\log u}})$, the expected running time $T(u)$ of `BuildTree` obeys the recursion

$$T(u) = O(u) + \sum_i T(m_i),$$

where

$$\sum_i m_i = O(u) \text{ and } \max_i m_i = O(u\sqrt{\log u}/2^{\sqrt{\log u}}).$$

Therefore, $T(u) = O(u\sqrt{\log u})$. Queries are answered by a recursive procedure called `Query`(P) (Algorithm 4). Step 1

Algorithm `Query`(P)
1) If $n \le 2^{\sqrt{\log u}}\sqrt{\log u}$, use a traditional algorithm to find $\operatorname{conv} P$ and return.
2) For $i = 1, \ldots, \ell - 1$
   a) Let $P_i = P \cap R_i$ and determine the intersections P_{ij} of P_i with the sets $R_i^{(j)}$.
   b) For all non-empty P_{ij}, call `Query`(P_{ij}) to compute $\operatorname{conv} P_{ij}$.
   c) Merge the $\operatorname{conv} P_{ij}$ into $\operatorname{conv} S_i$.
   d) Use `SplitHull` to extract $\operatorname{conv} P_i$ from the convex hull $\operatorname{conv}(P_i \cup S_i)$.
3) Let $P_\ell = U_\ell \cap P$. If $P_\ell \neq \emptyset$, call `Query`(P_ℓ) for $\operatorname{conv} P_\ell$.
4) Compute $\operatorname{conv} P$ by uniting $\operatorname{conv} P_1$, ..., $\operatorname{conv} P_\ell$.

Algorithm 4: Querying the simple scaffold tree.

takes $O\left(n\sqrt{\log u}\right)$ time [26]. With an appropriate pointer structure that provides links for the points in U to the corresponding subsets (as in the pointer-based implementation of vEB trees [36]), the total time for Step 2a is $O(n)$. The next claim handles Step 2c.

Claim 4.4. *Step 2c takes $O(|P_i|)$ time.*

Proof: Fix a j with $P_{ij} \neq \emptyset$. We show how to insert P_{ij} into $\operatorname{conv} S_i$ in time $O(|P_{ij}|)$. This implies the claim, since simple geometric arguments [20, Lemma 4.5] show that there can be no edge between two points $p \in P_{ij}$ and $q \in P_{ij'}$, for $j \neq j'$. Furthermore, the only facets in $\operatorname{conv} S_i$ that are destroyed are facets in $\Delta = \bigcup_{p \in P_{ij}} D_p$ by definition of D_p. By Lemma 4.2(2,3), the size of Δ is constant, because all the D_p have constant size, form connected components in the dual of $\operatorname{conv} S_i$ (a 3-regular graph), and have one facet in common. Thus, all we need to do is insert the constantly many points in S_i incident to a facet in Δ into $\operatorname{conv} P_{ij}$, which takes time $O(|P_{ij}|)$, as claimed. ■

By Theorem 4.1, Step 2d needs $O(|P_i|+|S_i|)$ time. Using an algorithm for merging convex hulls [19], Step 4 can be done in time $O(n \log \ell) = O(n \log\log u)$. Thus, the total time for Steps 2 to 4 is $O(n \log\log u + \sum_{i=1}^{\ell} |S_i|) = O(n \log\log u)$, by (3) and Step 1. Since there are $O(\sqrt{\log u})$ levels and since the computation in Step 1 is executed only once for each point in P, the result follows. ■

4.2. Bootstrapping the tree

We now describe the bootstrapping step. The main idea is to increase the degree in each node of the scaffold tree and to use a more basic tree to combine the results from the recursive calls quickly. However, if we are not careful, we could lose a constant factor in each bootstrapping step, which would not give the desired running time. To avoid this, we need the following result, which is similar to the

simple proof of Theorem 2.1 and whose proof we omit from this abstract.

Theorem 4.5. *Let $U \subseteq \mathbb{R}^3$ be in gcp, and let Let $S = \bigcup_{i=1}^{k} S_i \subseteq U$ with $|S| = m$, such that $|S_i| \leq c$, for some constant c, and such that the subgraphs $\operatorname{conv} U|_{S_i}$ are connected. Furthermore, let $S' \subseteq S$ be such that S' contains exactly one point from each S_i, chosen uniformly at random, and suppose $\operatorname{conv}(S')$ is available, and that we have a vEB structure for the neighbors of each vertex in $\operatorname{conv}(U)$. Then we can find $\operatorname{conv}(S)$ in expected time $O(m \log\log u)$.* □

Theorem 4.6. *Let $k > 1$ and $U \subseteq \mathbb{R}^3$ be a u-point set in gcp, and let $\operatorname{conv} U$ be given. Set $l_k = (\log u)^{1/k}$. There is a constant β with the following property: if $D_k(U)$ is a data structure for convex hull queries with preprocessing time $P_k(u) = O(ukl_k)$ and query time $Q_k(n,u) \leq \beta n k l_k \log\log u$, then there is a data structure $D_{k+1}(U)$ with preprocessing time $P_{k+1}(u) = O(u(k+1)l_{k+1})$ and query time $Q_{k+1}(n,u) \leq \beta n(k+1)l_{k+1}\log\log u$. The constants in the O-notation do not depend on k.*

Proof: Since the function $x \mapsto x \log^{1/x} u$ reaches its minimum for $x = \ln 2 \log\log u$, we may assume that $k < 0.7 \log\log u$, since otherwise the theorem holds by assumption. The preprocessing is very similar to Algorithm 3 with a few changes: (i) we iterate the loop in Step 2 while $|U_i| > u/2^{l_{k+1}^k}$; (ii) we apply Lemma 4.2 with $\alpha = 2^{(\log|U_i|)^{k/(k+1)}}$; and (iii) for each sample S_i we compute a data structure $D_k(S_i)$ for convex hull queries, which exists by assumption, for details, see Algorithm 5. Since the sizes

Algorithm $\texttt{BuildTree}_{k+1}(U)$
1) Let $U_1 = U$ and $i = 1$
2) While $|U_i| > u/2^{l_{k+1}^k}$.
   a) Apply Lemma 4.2 to U_i, where we set $\alpha = 2^{(\log|U_i|)^{k/(k+1)}}$. This yields subsets $S_i, R_i \subseteq U_i$, a partition $\mathcal{P}_i = \left(R_i^{(j)}\right)_{1\leq j\leq\beta}$ of R_i, and the convex hulls $\operatorname{conv} S_i$, $\operatorname{conv} R_i^{(j)}$ with the properties of Lemma 4.2.
   b) Execute $\texttt{BuildTree}_{k+1}\left(R_i^{(j)}\right)$ for $j = 1, \ldots, \beta$ and compute a data structure $D_k(S_i)$ for S_i.
   c) Let $U_{i+1} = U_i \setminus (S_i \cup R_i)$. Compute $\operatorname{conv} U_{i+1}$ using `SplitHull`, and create a vEB structure for the neighbors of each vertex. Increment i.
3) Let $\ell = i$ and call $\texttt{BuildTree}_{k+1}(U_\ell)$.

Algorithm 5: Bootstrapping the scaffold tree.

of the U_i decrease geometrically, we have $\ell = \Theta\left(l_{k+1}^k\right)$ and the total time for Step 2a is $O(u)$. The total time to construct the $D_k(S_i)$ in Step 2b is $\Theta\left(l_{k+1}^k\right) P_k\left(2^{l_{k+1}^k}\right) = O\left(\left(l_{k+1}^k\right) k 2^{l_{k+1}^k} l_{k+1}\right) = o(u)$, and similarly for the vEB trees in Step 2c. Since the sets $R_i^{(j+1)}$ have size $O(ul_{k+1}^k/2^{l_{k+1}^k})$, the number of levels is $O\left((k+1)l_{k+1}\right)$. The work at each level is $O(u)$, therefore the total preprocessing time is $P_{k+1}(u) = O\left(u(k+1)l_{k+1}\right)$. Queries are answered by $\texttt{Query}_{k+1}$ (Algorithm 6).

Algorithm $\texttt{Query}_{k+1}(P)$
1) For $i = 1, \ldots, \ell - 1$
   a) Let $P_i = P \cap R_i$ and determine the intersections P_{ij} of P_i with the sets $R_i^{(j)}$.
   b) For each non-empty P_{ij}, if $|P_{ij}| \leq \beta k l_{k+1}$, compute $\operatorname{conv} P_{ij}$ directly, else call $\texttt{Query}_{k+1}(P_{ij})$.
   c) For each nonempty P_{ij}, determine the set S_{ij} of points adjacent to a conflict facet of P_{ij}. Compute $\operatorname{conv}(P_{ij} \cup S_{ij})$.
   d) Let S_i' be a set that contains one random point from each S_{ij}. Use $D_k(S_i)$ to find $\operatorname{conv} S_i'$.
   e) Use Theorem 4.5 in order to compute the convex hull $\operatorname{conv}(P_i \cup \bigcup_j S_{ij})$.
   f) Use `SplitHull` to extract $\operatorname{conv} P_i$ from the convex hull $\operatorname{conv}(P_i \cup \bigcup_j S_{ij})$.
2) Recursively compute $\operatorname{conv} P_\ell$, where $P_\ell = U_\ell \cap P$.
3) Compute $\operatorname{conv} P$ by taking the union of $\operatorname{conv} P_1, \ldots, \operatorname{conv} P_\ell$.

Algorithm 6: Querying the bootstrapped scaffold tree.

In the following, let c denote a large enough constant. As before, Step 1a takes time cn with an appropriate pointer structure. In Step 1b, let n_1 denote the total number of points for which we compute the convex hull directly, and let $n_2 = n - n_1$. Then this step takes time $cn_1 \log\log u + cn_1 \log\beta + Q_{k+1}\left(n_2, cu\frac{l_{k+1}^k}{2^{l_{k+1}^k}}\right)$, assuming that Q_{k+1} is linear in the first and monotonic in the second component (which holds by induction on the second component). Since the conflict size of each P_{ij} is constant, Step 1c takes time cn, if we just insert the points in each S_{ij} into $\operatorname{conv} P_{ij}$ one by one. Furthermore, since we select one point per conflict set, the total size of the sets S_i' in Step 1d is at most $n_1 + n_2/(\beta k l_{k+1})$, so computing the convex hulls $\operatorname{conv} S_1', \ldots, \operatorname{conv} S_\ell'$ takes time

$$Q_k\left(n_1 + \frac{n_2}{\beta k l_{k+1}}, 2^{l_{k+1}^k}\right) \leq \beta k\left(n_1 + \frac{n_2}{\beta k l_{k+1}}\right) \cdot l_{k+1}\log\log u = (\beta n_1 k l_{k+1} + n_2)\log\log u.$$

By Theorems 4.5 and 4.1, Steps 1e and 1f take time $O(n\log\log u + n + \sum_i |S_i'|) \leq cn\log\log u$. Step 2 is already accounted for by Step 1b. Finally, Step 3 requires $cn\log\log u$ steps. By summing all the steps, we get the

following recursion for $Q_{k+1}(n,u)$:

$$Q_{k+1}(n,u) \le 5cn \log\log u + cn_1 \log\beta + \beta n_1 k l_{k+1} \log\log u + Q_{k+1}\left(n_2, cu\frac{l_{k+1}^k}{2^{l_{k+1}^k}}\right).$$

By induction, $Q_{k+1}(n,u) \le \beta n(k+1) l_{k+1} \log\log u$. ∎

Corollary 4.7. *Let $U \subseteq \mathbb{R}^3$ be a u-point set in gcp. In $O(u \log u)$ time, we can construct a data structure for convex hull queries with expected query time $O(n(\log\log u)^2)$. The space needed is $O(u(\log\log u)^2)$, and if $\operatorname{conv} U$ is available, preprocessing can also be done in this time.*

Proof: For $k = \frac{1}{2}\log\log u$, we have $(\log u)^{1/k} = O(1)$, and the result follows from Theorems 4.3 and 4.6. ∎

Corollary 4.8. *Let $P \subseteq \mathbb{R}^3$ be an n-point set in gcp and $c\colon P \to \{1,\ldots,\chi\}$ a coloring of P. Given $\operatorname{conv} P$, we can find $\operatorname{conv} c^{-1}(1), \ldots, \operatorname{conv} c^{-1}(\chi)$ in expected time $O(n(\log\log n)^2)$.* □

Acknowledgments

We would like to thank Bernard Chazelle, David Eppstein, Jeff Erickson, and Mikkel Thorup for stimulating and insightful discussions. K. Buchin was supported by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant no. 642.065.503 and project no. 639.022.707. W. Mulzer was supported in part by NSF grant CCF-0634958 and NSF CCF 083279.

References

[1] A. Aggarwal, "Lecture notes in computational geometry," *MIT Research Seminar Series MIT/LCS/RSS*, vol. 3, August 1988.

[2] S. Albers and T. Hagerup, "Improved parallel integer sorting without concurrent writing," *Inform. and Comput.*, vol. 136, no. 1, pp. 25–51, 1997.

[3] N. Amenta, D. Attali, and O. Devillers, "Complexity of Delaunay triangulation for points on lower-dimensional polyhedra," in *Proc. 18th SODA*, 2007, pp. 1106–1113.

[4] N. Amenta, S. Choi, and G. Rote, "Incremental constructions con BRIO," in *Proc. 19th SoCG*, 2003, pp. 211–219.

[5] A. Amir, A. Efrat, P. Indyk, and H. Samet, "Efficient regular data structures and algorithms for dilation, location, and proximity problems," *Algorithmica*, vol. 30, no. 2, pp. 164–187, 2001.

[6] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, "Sorting in linear time?" *JCSS*, vol. 57, no. 1, pp. 74–93, 1998.

[7] S. Arora and B. Barak, *Computational complexity: A Modern Approach*. Cambridge University Press, 2009.

[8] D. Attali and J.-D. Boissonnat, "A linear bound on the complexity of the Delaunay triangulation of points on polyhedral surfaces," *DCG*, vol. 31, no. 3, pp. 369–384, 2004.

[9] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conferences*. ACM Press, 1968, pp. 307–314.

[10] M. Bern, D. Eppstein, and J. Gilbert, "Provably good mesh generation," *JCSS*, vol. 48, no. 3, pp. 384–409, 1994.

[11] M. Bern, D. Eppstein, and S.-H. Teng, "Parallel construction of quadtrees and quality triangulations," *IJCGA*, vol. 9, no. 6, pp. 517–532, 1999.

[12] K. Buchin, "Organizing point sets: Space-filling curves, Delaunay tessellations of random point sets, and flow complexes," Ph.D. dissertation, Free University Berlin, 2007, http://www.diss.fu-berlin.de/diss/receive/FUDISS_thesis_000000003494.

[13] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *JACM*, vol. 42, no. 1, pp. 67–90, 1995.

[14] T. M. Chan, "Closest-point problems simplified on the RAM," in *Proc. 13th SODA*, 2002, pp. 472–473.

[15] ——, "Well-separated pair decomposition in linear time?" *IPL*, vol. 107, no. 5, pp. 138–141, 2008.

[16] T. M. Chan and M. Pătraşcu, "Transdichotomous results in computational geometry, I: Point location in sublogarithmic time," sICOMP To appear.

[17] T. M. Chan and M. Pătraşcu, "Voronoi diagrams in $n2^{O(\sqrt{\lg\lg n})}$ time," in *Proc. 39th STOC*, 2007, pp. 31–39.

[18] B. Chazelle, O. Devillers, F. Hurtado, M. Mora, V. Sacristán, and M. Teillaud, "Splitting a Delaunay triangulation in linear time," *Algorithmica*, vol. 34, no. 1, pp. 39–46, 2002.

[19] B. Chazelle, "An optimal algorithm for intersecting three-dimensional convex polyhedra," *SICOMP*, vol. 21, no. 4, pp. 671–696, 1992.

[20] B. Chazelle and W. Mulzer, "Computing hereditary convex structures," in *Proc. 25th SoCG*, 2009, pp. 61–70.

[21] L. P. Chew and S. Fortune, "Sorting helps for Voronoi diagrams," *Algorithmica*, vol. 18, no. 2, pp. 217–228, 1997.

[22] K. L. Clarkson, "Fast algorithms for the all nearest neighbors problem," in *Proc. 24th FOCS*, 1983, pp. 226–232.

[23] K. L. Clarkson and C. Seshadhri, "Self-improving algorithms for Delaunay triangulations," in *Proc. 24th SoCG*, 2008, pp. 148–155.

[24] K. L. Clarkson and P. W. Shor, "Applications of random sampling in computational geometry. II," *DCG*, vol. 4, no. 5, pp. 387–421, 1989.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[26] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008.

[27] M. de Berg, M. van Kreveld, and J. Snoeyink, "Two- and three-dimensional point location in rectangular subdivisions," *J. Algorithms*, vol. 18, no. 2, pp. 256–277, 1995.

[28] H. N. Djidjev and A. Lingas, "On computing Voronoi diagrams for sorted point sets," *IJCGA*, vol. 5, no. 3, pp. 327–337, 1995.

[29] M. L. Fredman, "How good is the information theory bound in sorting?" *TCS*, vol. 1, no. 4, pp. 355–361, 1975/76.

[30] M. L. Fredman and D. E. Willard, "Surpassing the information theoretic bound with fusion trees," *JCSS*, vol. 47, no. 3, pp. 424–436, 1993.

[31] Y. Han and M. Thorup, "Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space," in *Proc. 43rd FOCS*, 2002, pp. 135–144.

[32] J. Iacono and S. Langerman, "Dynamic point location in fat hyperrectangles with integer coordinates," in *Proc. 12th*

CCCG, 2000, pp. 181–186.

[33] R. G. Karlsson and M. H. Overmars, "Scanline algorithms on a grid," *BIT*, vol. 28, no. 2, pp. 227–241, 1988.

[34] D. Kirkpatrick and S. Reisch, "Upper bounds for sorting integers on random access machines," *TCS*, vol. 28, no. 3, pp. 263–276, 1984.

[35] J. v. Leeuwen and A. Tsakalides, "An optimal pointer machine algorithm for finding nearest common ancestors," Department of Information and Computing Sciences, Utrecht University, Tech. Rep. RUU-CS-88-17, 1988.

[36] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1984, vol. 1.

[37] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis, "Separators for sphere-packings and nearest neighbor graphs," *JACM*, vol. 44, no. 1, pp. 1–29, 1997.

[38] R. Motwani and P. Raghavan, *Randomized algorithms*. Cambridge University Press, 1995.

[39] K. Mulmuley, *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, 1994.

[40] M. H. Overmars, "Computational geometry on a grid: An overview," Rijksuniversiteit Utrecht, Tech. Rep. RUU-CS-87-04, 1987.

[41] A. Schönhage, "On the power of random access machines," in *Proc. 6th ICALP*, 1979, pp. 520–529.

[42] R. Seidel, "A method for proving lower bounds for certain geometric problems," Cornell University, Ithaca, NY, USA, Tech. Rep. TR84-592, 1984.

[43] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Math. Systems Theory*, vol. 10, no. 2, pp. 99–127, 1976/77.

[44] M. J. van Kreveld, M. Löffler, and J. S. B. Mitchell, "Preprocessing imprecise points and splitting triangulations," in *Proc. 19th ISAAC*, 2008, pp. 544–555.

[45] D. E. Willard, "Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree," *SICOMP*, vol. 29, no. 3, pp. 1030–1049, 2000.

APPENDIX A.
SHUFFLE-SORTING ON A WORD RAM

We show how to shuffle-sort a set of points $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^d$ in expected time $O(n \log\log n)$ on a standard word RAM, avoiding the shuffle operation. For this, we adapt a sorting algorithm by Andersson *et al* [6]. Let $w \geq \log n$ be the word size and b the bit size for the coordinates of the p_i (ie, $p_i \in [0, 2^b - 1]^d$). We assume that w and b are known. First, we extend a result by Albers and Hagerup [2] which concerns large word sizes.

Theorem A.1. *Suppose $b \leq \lceil w/(d \log n \log\log n) \rceil - 1$. Then P can be sorted according to the shuffle order in $O(n)$ time.*

Proof: We can fit $2k = w/(db+1) \geq \log n \log\log n$ *fields* in one word, each containing a point and a *testbit*. For a word a, let $a[i]$ be its i-th field, and $a[i].t, a[i].d$ its test bit and data. We need a procedure for merging two sorted words quickly [2, Section 3].

Claim A.2. *Given two words w_1, w_2 containing two sorted sequences $(p_i), (q_i)$ of k points each, we can compute a word w' containing the merged sequence (z_i) in time $O(\log k)$.*

Proof: We use a bitonic sorting network by Batcher [9], which requires the following subroutine: given two words a and b, each containing k points, compute in constant time a word z such that[3] $z[i].t = [a[i].d <_\sigma b[i].d]$ for each i. We use an algorithm

Algorithm `BatchCompare`

1) Create d copies $a_1, \ldots, a_d$ and $b_1, \ldots, b_d$ of a and b.
2) Shift and mask the words a_j, b_j so that $a_j[i].d = p_{ij}$ and $b_j[i].d = q_{ij}$ for $i = 1, \ldots, k$, where p_{ij} and q_{ij} denote the jth coordinates of p_i and q_i.
3) Create d words $c_1, \ldots, c_d$ with $c_j[i].d = a_j[i].d \oplus b_j[i].d$ for $i = 1, \ldots, k$, where $\oplus$ means bitwise XOR.
4) Create words g, h such that $g = c_1$ and $h = (1, \ldots, 1)$, ie, the word with a 1 in the data item of each field.
5) For $j = 2, \ldots, d$:
   a) Set the testbits in c_j to $c_j[i].t = [g[i].d \leq c_j[i].d]$ and let M be a mask for the fields i st $c_j[i].t = 1$.
   b) Let $c_j = c_j \oplus (g \texttt{ AND } M)$.
   c) Set the testbits in c_j to $c_j[i].t = [g[i].d < c_j[i].d]$ and compute a mask M' for the fields i with $c_j[i].t = 1$. Let $M = M \texttt{ AND } M'$.
   d) Set $g = (g \texttt{ AND } \overline{M}) \texttt{ OR } ((a_j \oplus b_j) \texttt{ AND } M)$, where $\overline{M}$ is the bitwise negation of M. Set $h = (h \texttt{ AND } \overline{M}) \texttt{OR } ((j, \ldots, j) \texttt{AND } M)$.
6) Create a word z with $z[i].t = [a_{h[i]}[i] < b_{h[i]}[i]]$.

Algorithm 7: Comparing many points simultaneously.

`BatchCompare`, based on a technique by Chan [14]. For a b-bit integer x, let $|x| = \lfloor \log x \rfloor$ for $x \neq 0$ and $|0| = 1$, ie, $|x|$ is the position of the most significant bit in $x \neq 0$. For each i, `BatchCompare` finds the smallest coordinate $h[i]$ that has maximum $|c_{h[i]}[i]|$. Then, it determines whether $p_i <_\sigma q_i$ by comparing their coordinate $h[i]$. Correctness follows immediately from the definition of the shuffle order. The main observation is that a comparison $|x| < |y|$ can be done as

$$[|x| < |y|] = [(x \leq y) \wedge (x < x \oplus y)]. \tag{4}$$

`BatchCompare` runs in constant time, assuming that the constants $(1, \ldots, 1), \ldots, (d, \ldots, d)$ have been precomputed, which takes $O(\log k)$ time. In particular, note that Step 6 takes constant time since there are only constantly many possible indices $h[i]$. ■
Given Claim A.2, the theorem follows using merge sort [2]. ■

Next, we need to reduce the number of bits per point:

Theorem A.3. *With expected linear time and space overhead, the problem of shuffle-sorting n points with b-bit coordinates can be reduced to sorting n points with $b/2$-bit coordinates.*

Proof: The proof is verbatim as in Kirkpatrick and Reisch [34, Section 4]: bucket the points by the upper $b/2$ bits of their coordinates. By universal hashing, this takes linear expected time and space. From each nonempty bucket b, select its maximum m_b. Truncate each m_b to the upper $b/2$ bits and store a flag with its satellite data. The remaining points are truncated to the lower $b/2$ bits, and the corresponding bucket is stored in the satellite data. After sorting the resulting point set, we can establish (i) the ordering of the buckets, using the m_b, and (ii) the ordering within each bucket. The crucial fact is that the for any two points p, q, we have $p <_\sigma q$ precisely if $(p' <_\sigma q') \vee (p' = q' \wedge p'' <_\sigma q'')$, where (p', p''), (q', q'') are derived from p, q by splitting each of their coordinates into two blocks of $b/2$ bits. ■

As in [6] combining Theorems A.3 and A.1, yields

Theorem A.4. *An n-point set with b-bit coordinates can be shuffle-sorted in expected time $O(n \log\log n)$ with $O(n)$ space.* □

[3] We use Iverson's notation: $[X] = 1$ if X is true and $[X] = 0$, otherwise.