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

KEVIN BUCHIN
TU Eindhoven
and
WOLFGANG MULZER
Freie Universität Berlin

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 $O(\texttt{sort}(n))$ on a word RAM, where $\texttt{sort}(n)$ 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 expected 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 expected time $O(|P|(\log\log |U|)^2)$; (v) given a convex polytope in 3-space with n vertices which are colored with $\chi \geq 2$ colors, we can split it into the convex hulls of the individual color classes in expected time $O(n(\log\log n)^2)$.

The results (i)–(iii) generalize to higher dimensions, where the expected running time now also depends on the complexity of the resulting DT. 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.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Convex hull, Delaunay triangulation, Word RAM

1. INTRODUCTION

Everyone knows that it takes $\Omega(n \log n)$ time to sort n numbers (in a comparison model)—and yet this lower bound can often be beaten. Under the right assumptions, radix sort and bucket sort run in linear time [Cormen et al. 2009]. Using van Emde Boas (vEB) trees [van Emde Boas 1977; van Emde Boas et al. 1976], 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 expected time $O(n\sqrt{\log n})$ (without randomization, the running time is $O(n \log n / \log\log n)$). Fusion trees were introduced in 1990 by Fredman and Willard [1993] and triggered off a development (see, for example, [Andersson et al. 1998; Raman 1996; Han 2004; Han and Thorup 2002; Thorup 1998]) that culminated in the $O(n\sqrt{\log\log n})$ expected time integer sorting algorithm by Han and Thorup [2002]. For small and large word sizes (that is, for word size $w = \Theta(\log n)$ or $w = \Omega(\log^{2+\varepsilon} n)$), we can even sort in linear time (via radix sort [Cormen et al. 2009] or signature sort [Andersson et al. 1998], respectively).

In computational geometry, there have been many results that use vEB trees or similar structures to surpass traditional lower bounds (e.g., [Amir et al. 2001; de Berg et al. 1995; Chew and Fortune 1997; Iacono and Langerman 2000; Karlsson 1985; Karlsson and Overmars 1988; Overmars 1987]). 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 [2000] applied

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 and a Wallace Memorial Fellowship in Engineering.

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, an inherently non-orthogonal structure. The breakthrough came in 2006, when Chan and Pătraşcu [2009] discovered transdichotomous algorithms for point location in non-orthogonal planar subdivisions. This led to better bounds for many classic computational geometry problems. In a follow-up paper [Chan and Pătraşcu 2007], 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 the merits of various approaches to transdichotomous algorithms and why fusion trees seemed most feasible for planar point location [Chan and Pătraşcu 2009, Section 3]. 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 quickly, for example a pointer machine [Mehlhorn 1984]. This becomes particularly useful for *hereditary* results [Chazelle and Mulzer 2009a], 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 given, large enough, subset of U faster than in $\Theta(n \log n)$ time, and we can ask to what extent similar results are possible for problems other than sorting.

Coming back to geometric problems, there are many incremental algorithms for the Delaunay triangulation that use orthogonal data structures (such as quadtrees or kd-trees) [Amenta et al. 2003; Bentley et al. 1980; Isenburg et al. 2006; Liu and Snoeyink 2005; Ohya et al. 1984a; 1984b; Su and Drysdale 1997; Zhou and Jones 2005], and they perform well in experiments (and on random points). Empirically, the overhead for constructing, say, a quadtree is negligible compared to the time for the Delaunay triangulation, even though there is no difference asymptotically. The reason for the good results in practice is that typically the running time is dominated by the point location cost, which can often be drastically reduced with an orthogonal data structure. However, in the worst case all these heuristics need quadratic time (unless we use an optimal point location structure as a fall-back).

Some algorithms bucket the points into an $\sqrt{n} \times \sqrt{n}$ grid [Bentley et al. 1980; Su and Drysdale 1997], so that they can then be located by spiral search. However, if too many points end up in one grid cell, we get quadratic running time. Other heuristics insert the points according to a space-filling curve corresponding to a depth-first traversal of a quadtree (see Figure 1 for the so-called Morton curve or Z-order). But again, this insertion order cannot avoid quadratic worst-case behavior, even in the plane [Buchin 2009] (without an additional point location structure). Traversal orders of kd-trees have also been used [Amenta et al. 2003]. In the light of these algorithms, it is natural to ask whether we can exploit ideas from these heuristics to design better worst-case optimal algorithms.

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 [Amenta et al. 2003; Clarkson and Shor 1989; Mulmuley 1994] that relies on *dependent* sampling for faster conflict location (see also [Chazelle and Mulzer 2009b] for another look at RICs with dependencies in the underlying randomness). 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, e.g., point sets suitably sampled from a $(d-1)$-dimensional polyhedron in $\mathbb{R}^d$ [Amenta et al. 2007; Attali and Boissonnat 2004].[1] The algorithm is relatively simple

[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

and works in any dimension, but the analysis turns out to be rather subtle.[2] It is a well-known fact that given a quadtree for a point set, its nearest-neighbor graph can be computed in linear time [Callahan and Kosaraju 1995; Chan 2008; Clarkson 1983]. 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 expected time proportional to the expected structural change of an 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. Using this connection between quadtrees and DTs, we obtain several results.

—**DTs on a word RAM.** 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 expected time $O(\mathtt{sort}(n))$ on a word RAM. By $\mathtt{sort}(n)$ we denote the time to sort n numbers, which is at most $O(n\sqrt{\log\log n})$ if randomization is allowed [Han and Thorup 2002]. In case of word size $\Theta(\log n)$ or $\Omega(\log^{2+\varepsilon} n)$, the expected running time of our algorithm is $O(n)$ using radix-sort [Cormen et al. 2009] or signature sort [Andersson et al. 1998], respectively. 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 exemplify this by showing how to adapt the (comparatively simple) $O(n \log\log n)$ sorting algorithm by Andersson et al. [1998].

—**DTs from a fixed universe.** We can preprocess a point set $U \subseteq \mathbb{R}^d$ such that for any subset $P \subseteq U$ it takes $O(|P| \log\log |U| + C(P))$ expected time to find $\mathrm{DT}(P)$. Here, $C(P)$ denotes the expected structural change of an RIC on P.

—**DTs for presorted point sets.** Since a planar quadtree can be computed by an algebraic computation tree (ACT) [Arora and Barak 2009, Chapter 16.2] 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 [Djidjev and Lingas 1995], and also for convex hulls in $\mathbb{R}^3$ when the points are sorted in any constant number of directions [Seidel 1984]. This problem has appeared in the literature for at least twenty years [Aggarwal 1988; Djidjev and Lingas 1995; Chew and Fortune 1997]. Our result seems to mark the first non-trivial progress on this question, and it shows that unlike for convex hulls and point sets sorted in one direction, a Ben-Or style lower bound in the algebraic decision tree model [Ben-Or 1983] does not exist. However, we do not know if a quadtree for presorted points can indeed be constructed in linear time, since the algorithms we know still need an $\Omega(n \log n)$ overhead for data handling. It would be interesting to see if there is a connection to the notorious Sorting $X + Y$ problem [Fredman 1976], which seems to exhibit a similar behavior.

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 expected time $O(|P|(\log\log |U|)^2)$. These queries are called *convex hull queries*. We use a relatively recent technique [Chazelle and Mulzer 2009a; Clarkson and Seshadhri 2008; van Kreveld et al. 2008] 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 [Chazelle and Mulzer 2009a] to remove the scaffold. We also need a carefully balanced recursion and a bootstrapping method similar to the one by Chan and Pătraşcu [2007]. This also improves a recent algorithm for splitting a 3-polytope whose vertices are colored with $\chi \geq 2$ colors into its monochromatic parts [Chazelle and Mulzer 2009a, Theorem 4.1]: now we can achieve an expected running time of $O(n(\log\log n)^2)$ instead of $O(n\sqrt{\log n})$.

All our algorithms are randomized. In subsequent work, Löffler and Mulzer [2011] gave a deterministic way to find the Delaunay triangulation of a planar point set P in linear time, given a quadtree for P. This yields deterministic versions for many of the algorithms we present here. In particular, the result by Löffler and Mulzer [2011] shows that on a word RAM Delaunay triangulations can be computed deterministically in time $O(n \log\log n)$ [Han 2004]. Nonetheless, their results do not extend to higher dimensions. In particular,

structural change of an RIC.

[2]At least for $d > 2$. For the planar case, the analysis can be simplified considerably, see below.

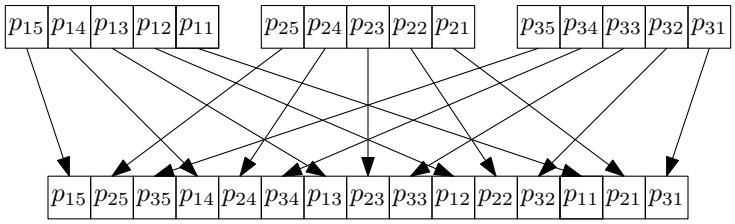

Fig. 1. The shuffle operation for a 3-dimensional point with 5-bit coordinates.

it would still be very interesting to find a deterministic algorithm for splitting convex polytopes in three dimensions [Chazelle and Mulzer 2009a].

Computational models. We use two different models of computation, a *word RAM* and a *pointer machine*. The word RAM represents the data as a sequence of w-bit words, where $w \geq \log n$. Data can be accessed randomly, and standard operations, such as Boolean operations, addition, or multiplication take constant time. We will also 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 `shuffle`(p) is the dw-bit word $p_{1w} p_{2w} \dots p_{dw} \dots p_{12} \dots p_{d2} p_{11} \dots p_{d1}$; see Figure 1. Clearly, `shuffle` is in AC^0, and we assume that it takes constant time on our RAM. In Appendix A, we shall explain 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 [de Berg et al. 2008, 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 [Schönhage 1979].

2. FROM NEAREST-NEIGHBOR GRAPHS TO DELAUNAY TRIANGULATIONS

We now describe our reduction from Delaunay triangulations (DTs) to nearest-neighbor graphs (NNGs), for a d-dimensional point set P. This is done by a randomized algorithm that we call `BrioDC`, an acronym for **B**iased **R**andom **I**nsertion **O**rder with **D**ependent **C**hoices; see Algorithm 1.

Algorithm 1 The reduction from Delaunay triangulation to nearest-neighbor graph.

`BrioDC`(P)

(1) If $|P| = O(1)$, compute $\mathrm{DT}(P)$ directly and return.
(2) Compute $\mathrm{NN}(P)$, the nearest-neighbor graph for 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.

In the following we assume that the reader is familiar with the standard randomized incremental construction of Delaunay triangulations [de Berg et al. 2008; Mulmuley 1994]. 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 the simplex containing p in the current DT, and insert p by performing the appropriate edge flips [de Berg et al. 2008; Mulmuley 1994]. Repeat until all of P has been processed.

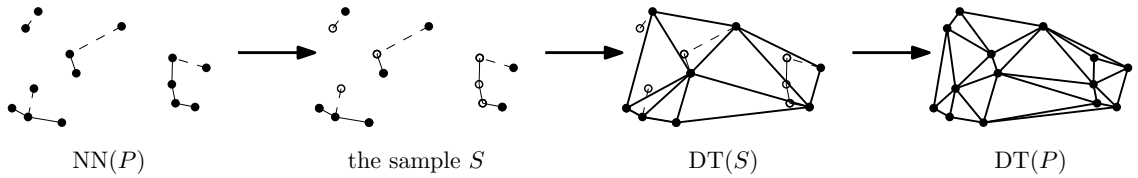

Fig. 2. The algorithm `BrioDC`: the edges of $\mathcal{M}(P)$ are shown dashed, the remaining edges of NN(P) are solid.

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 monotonically 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 an RIC on P. The constant in the O-notation depends exponentially on d.*

Before proving Theorem 2.1 in general, we give a simple proof of the theorem for the case that P is planar (which implies that $C(P) = O(n)$). For the planar case, we later (Remark 3.6) also give an alternative linear-time reduction from DTs to quadtrees that uses a combination of known results.

PROOF OF THEOREM 2.1 FOR $d = 2$. Since $f(m)/m$ increases, the expected cost to compute the NNGs for all the samples is $O(f(n))$. Furthermore, the cost of tracing an edge pq of 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 NN(P), which is constant [Miller et al. 1997, 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. Note that the argument above also holds for $d > 2$.

We now use induction to show that there exists a constant c such that the expected structural change is at most cn. This clearly holds for $n = O(1)$. Now we suppose that DT(S) can be found with expected structural change at most $c|S|$, and we examine what happens in Step 5 of `BrioDC`. Let t be a triangle defined by three points of P, and let B_t denote the points of P inside the circumcircle of t. The points in B_t are called *stoppers*, and the size of B_t is referred to as the *conflict size* for t. The triangle t can only appear in Step 5 if $S \cap B_t = \emptyset$. Now let p_s be the probability that a given triangle t with conflict size s appears in Step 5. Clearly, we have $p_s \leq 1/2^s$: if the stoppers of t are sampled independently of each other, we directly get this bound, and otherwise S includes a stopper and t cannot be created at all. By the well-known Clarkson-Shor bound [Clarkson and Shor 1989], the number of triangles with conflict size at most s is $O(ns^2)$, so the expected number of triangles created in Step 5 is $O\left(\sum_{s=0}^{\infty} ns^2/2^s\right) \leq dn$, for some constant d. Since the expected size of S is $n/2$, the total expected structural change is therefore $cn/2 + dn \leq cn$, for c large enough, which completes the induction. Note that this argument does not help in higher dimensions, because there the Clarkson-Shor bound gives $O(n^{\lfloor (d+1)/2 \rfloor} s^{\lceil (d+1)/2 \rceil})$ simplices with conflict size at most s, which might yield a bound that is much larger than the expected structural change of an RIC. □

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

LEMMA 2.2. *We have*

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

Since Lemma 2.2 constitutes the most technical part of our proof, let us first provide some intuition. We visualize the sampling process as follows [Motwani and Raghavan 1995, Chapter 1.4]: imagine a particle that moves at discrete time steps on the nonnegative x-axis and always occupies integer points. Refer to Figure 3. The particle starts at position $|B_{\mathbf{u}}|$, and after β steps, it is at position $|S_\beta \cap B_{\mathbf{u}}|$, the number of stoppers in the current sample. Since we are looking to bound $\Pr[A_\alpha]$, the goal is to upper-bound the probability of reaching 0 in $\alpha+1$ steps while retaining all triggers $\mathbf{u}$. 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 sampled independently, but dependencies can only help, because in each stopper-stopper pair one stopper is guaranteed to survive. Eliminating $d+1$ stoppers in the i-th 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.

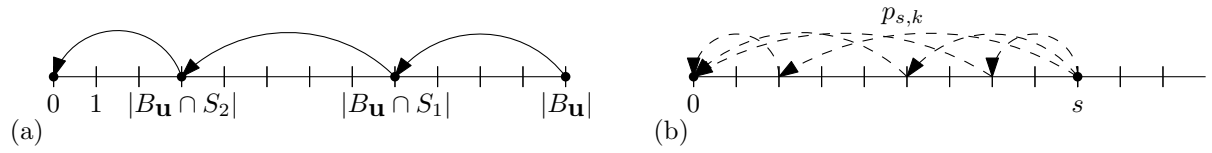

Fig. 3. (a) We visualize the sampling process as a particle moving from on the positive x-axis from $|B_{\mathbf{u}}|$ towards 0; (b) $p_{s,k}$ roughly corresponds to the probability of reaching 0 from s in k steps, while retaining all the triggers. We use appropriate conditioning in order to deal with the dependencies between the different events.

Proof of Lemma 2.2. For $S \subseteq P$, let the *matching profile* for S be the triple $(a,b,c) \in \mathbb{N}_0^3$ that counts the number of trigger-stopper, stopper-stopper, and trigger-trigger pairs in $\mathcal{M}(S)$. For a given α we consider

$$p_{s,k} = \max_{\mathcal{P}_k} \Pr[A_\alpha \mid X_{s,k}, \mathcal{P}_k], \tag{1}$$

where s and $k \le \alpha$ are integers and $X_{s,k} = \{\mathbf{u} \subseteq S_{\alpha-k}\} \cap \{|B_{\mathbf{u}} \cap S_{\alpha-k}| = s\}$ is the event that the sample $S_{\alpha-k}$ contains all triggers and exactly s stoppers. The maximum in (1) 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 $\Pr[A_\alpha] = p_{|B_{\mathbf{u}}|,\alpha}$, it suffices to upper-bound $p_{s,k}$. 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 $U_{i,k} = \{|B_{\mathbf{u}} \cap S_{\alpha-k}| = i\}$ denote the event that $S_{\alpha-k}$ contains exactly i stoppers (note that $X_{s,k} = T_k \cap U_{s,k}$).

Proposition 2.3. *We have*

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

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

Proof. Fix a sequence $\mathcal{P}_k$ as in (1). 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

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

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 $\mathbf{m}$. $\square$

We use Proposition 2.3 to bound $p_{s,k}$: if $\mathbf{m} = (a, b, c)$ pairs up two triggers (i.e., $c > 0$), 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

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

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. For the inductive step, by (2),

$$p_{s,k+1} \leq \max_{\substack{\mathbf{m} \\ c=0}} \sum_{i=b}^{s-a-b} \frac{p_{i,k}}{2^{d+1}} \Pr\left[B_{1/2}^{t_{\mathbf{m}}} = i - b\right] = \max_{\substack{\mathbf{m} \\ c=0}} \frac{1}{2^{d+1+t_{\mathbf{m}}}} \sum_{i=0}^{t_{\mathbf{m}}} \binom{t_{\mathbf{m}}}{i} p_{i+b,k}. \tag{3}$$

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 (3) 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. $\square$

Now, since $1 - x \geq \exp(-2x)$ for $0 \leq x \leq 1/2$, we have

$$\prod_{j=1}^{k}(1-2^{-j})^{-d-1} \leq \exp\left(2(d+1)\sum_{j=1}^{\infty}2^{-j}\right) = e^{2(d+1)},$$

so we get by Proposition 2.4

$$\Pr[A_\alpha] \leq p_{|B_\mathbf{u}|,\alpha} \leq e^{2d+2}2^{-(d+1)\alpha}(1-2^{-\alpha-1})^{|B_\mathbf{u}|},$$

which proves Lemma 2.2. □

The following lemma was proven by Buchin [2007, Lemma 3.5], and it allows us to relate the $\Pr[A_\alpha]$'s to the probability that the simplex Δ appears in a traditional RIC. For completeness we include a short proof.

LEMMA 2.5. *Let p_R denote the probability that the simplex spanned by $\mathbf{u} \subseteq P$ occurs in an RIC and let $p_\alpha = 2^{-(d+1)\alpha}(1-2^{-d-1})(1-2^{-\alpha-1})^{|B_u|}$. Then $\sum_{\alpha=0}^{\infty} p_\alpha \leq 2^{d+1}p_R$.*

PROOF. Consider the following way to obtain a random permutation of the points: for each $p \in P$ randomly pick a positive integer z_p, where $\Pr[z_p = \alpha] = 2^{-\alpha}$ for $\alpha > 0$ and all the z_p are drawn independently. Then subdivide the points into groups according to their label z_p, choose a random permutation in each group, and concatenate the groups in decreasing order according to their label. Note that $\Pr[z_p \geq \alpha] = 2^{-(\alpha-1)}$ for $p \in P$ and $\Pr[\{z_p \geq \alpha \mid p \in P'\}] = 2^{-|P'|(\alpha-1)}$ for $P' \subseteq P$.

For a simplex spanned by $\mathbf{u} \subseteq P$ we derive a lower bound on the probability that its last trigger (in the random permutation) has label $\alpha_0 > 0$ and that the simplex occurs in an RIC. A sufficient condition for this event is that the last trigger u has $z_u = \alpha_0$ (which happens with probability $\Pr[\{z_u \geq \alpha_0 \mid u \in \mathbf{u}\}] - \Pr[\{z_u \geq \alpha_0+1 \mid u \in \mathbf{u}\}] = 2^{-(d+1)(\alpha_0-1)}(1-2^{-d-1})$) and that all stoppers $q \in B_\mathbf{u}$ have $z_q < \alpha_0$ (which independently happens with probability $(1-2^{-\alpha_0})^{|B_\mathbf{u}|}$). A sufficient condition gives us a lower bound for an event, thus summing over α we get

$$p_R \geq \sum_{\alpha=1}^{\infty} 2^{-(d+1)\alpha}(1-2^{-d-1})(1-2^{-\alpha})^{|B_\mathbf{u}|} = 2^{-(d+1)}\sum_{\alpha=0}^{\infty} p_\alpha.$$

□

PROOF OF THEOREM 2.1. As we argued at the beginning of the proof for the planar case, it suffices to bound the structural change. It therefore is sufficient to show that the probability that the simplex spanned by $\mathbf{u} \subseteq P$ occurs in `BrioDC` is asymptotically upper-bounded by the corresponding probability in an RIC. In the case of `BrioDC`, this probability is bounded by $\sum_{\alpha=0}^{\infty}\Pr[A_\alpha]$. By Lemmas 2.2 and 2.5 we have

$$\sum_{\alpha=0}^{\infty}\Pr[A_\alpha] \leq e^{2d+2}\sum_{\alpha=0}^{\infty}2^{-(d+1)\alpha}\left(1-2^{-\alpha-1}\right)^{|B_\mathbf{u}|} \leq \frac{e^{2d+2}2^{d+1}}{1-2^{-d-1}}\,p_R,$$

where p_R denotes the probability that the simplex occurs in an RIC. □

Remark 2.6. 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 [Djidjev and Lingas 1995].

Remark 2.7. The dependent sampling has more advantages than just allowing for fast point location. For instance, assume P samples a region, e.g., a surface [Attali and Boissonnat 2004], in the sense that for any point in the region there is a point in P at distance at most ε. Now, the length of edges in the NNG is bounded by 2ε, and S_1 (if it is not empty) contains for any $p \in P$ a point from the 2-neighborhood of p in the NNG. Therefore, also S_1 samples the region with $\varepsilon_1 \leq 4\varepsilon$, or more generally for $i > 1$, S_i samples the region with $\varepsilon_i \leq 4^i\varepsilon$ if S_i is not empty.

Remark 2.8. Lemma 2.2 directly extends to the more general setting of *configuration spaces* [Mulmuley 1994] if we replace $d+1$ by the degree bound, i.e., 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 2.9. Buchin [2008] gave an alternative reduction from DTs to nearest-neighbor graphs that is also based on a randomized incremental construction, but with unbiased sampling. Like `BrioDC`, his algorithm uses a sequence $P = S_0 \supseteq \cdots \supseteq S_\ell$ of samples. But since the algorithm uses unbiased sampling, S_{i+1} does not necessarily meet every component of $\mathrm{NN}(S_i)$ in contrast to `BrioDC`. For the remaining components the algorithm uses a combination of nearest neighbor graphs of smaller subsets and the *history* structure for conflict location. Thus, the algorithm by Buchin [2008] is somewhat more involved, but the analysis is a little less technical (at least for $d > 2$).

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 also known as the Morton-order [Morton 1966] or the Z-order, and it is intimately related to quadtrees [Bern et al. 1999; Chan 2008]; see Figure 4.

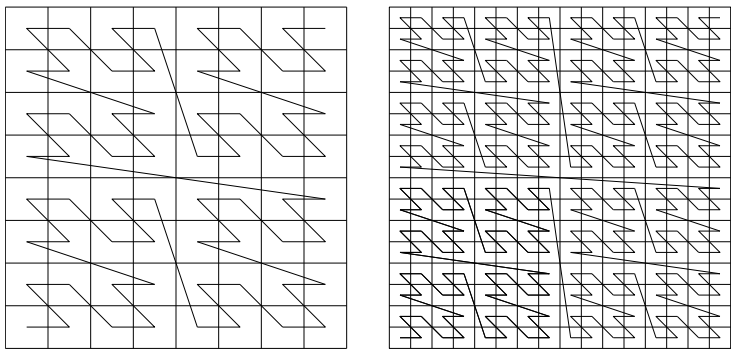

Fig. 4. The shuffle order corresponds to the left-to-right traversal of the leaves of a quadtree.

LEMMA 3.1. *Suppose our computational model is a word RAM, and that $P \subseteq \{0, \ldots, 2^w - 1\}^d$ is 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 [Chan 2008, Step 2]. We define a hierarchy H of quadtree boxes, by taking the hypercube $\{0, \ldots, 2^w - 1\}^d$ 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 compressed subtree of H induced by the leaves in H that correspond to the points in P (obtained by contracting all paths that consist of nodes with only one child). 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`; see Algorithm 2. The algorithm is similar to the construction of *Cartesian*

Algorithm 2 Building a compressed quadtree.

`BuildQuadTree`

(1) $q_0.\mathrm{box} = \{0, \ldots, 2^w - 1\}^d$, $q_0.\mathrm{children} = (p_1)$, $k = 0$
(2) for $i = 2, \ldots, n$
  (a) while $|\texttt{box}(p_{i-1}, p_i)| > |q_k.\mathrm{box}|$ do $k = k - 1$
  (b) if $|q_k.\mathrm{box}| = |\texttt{box}(p_{i-1}, p_i)|$, let p_i be the next child of q_k; otherwise, create q_{k+1} with $q_{k+1}.\mathrm{box} = \texttt{box}(p_{i-1}, p_i)$, 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 $k = k + 1$.

trees [Gabow et al. 1984]. Assuming our model supports the `msb` (most significant bit) operation[3] in constant time, the algorithm runs in linear time: the total number of iterations in Step 2a is bounded by the number of times k is decremented, which is clearly at most n, and the box sizes can be computed in constant time.

With some more effort, we can avoid the `msb` operation. First, as observed by Chan [2002], note that box sizes can be compared without the need for `msb`, because for two binary numbers x, y, we have

$$\texttt{msb}(x) < \texttt{msb}(y) \iff (x \le y) \wedge (x < x \oplus y), \tag{4}$$

where $x \oplus y$ denotes the bitwise `xor` operation. Thus, `BuildQuadTree` can find the combinatorial structure of the compressed quadtree T for P in linear time. Using a postorder traversal of T, we can find for each node b in T a minimum bounding box for the points under b, again in linear time. This information suffices to apply Lemma 3.2 below, as we can see by inspecting the proof of Callahan and Kosaraju [1995]. □

Via well-separated pair decompositions, we can go from quadtrees to NNGs in linear time as was shown by Callahan and Kosaraju [1995] and by Clarkson [1983] (see also [Chan 2008]). Their result is stated as follows:

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 Lemma 3.2, we establish a connection between quadtrees and Delaunay triangulations.

THEOREM 3.3. *Let $P \subseteq \mathbb{R}^d$. Given a compressed quadtree for P, we can find* $\mathrm{DT}(P)$ *in expected time $O(n + C(P))$, where $C(P)$ denotes the structural change of an RIC on P.*

PROOF. `BrioDC` from Theorem 2.1 generates a sequence of samples $P = S_0 \supseteq S_1 \supseteq \cdots \supseteq S_\ell$, and we need to find the nearest-neighbor graphs $\mathrm{NN}(S_0), \ldots, \mathrm{NN}(S_\ell)$ in total $O(n)$ time. By Lemma 3.2, and the fact that $\mathbf{E}\left[\sum_i |S_i|\right] = O(n)$, it suffices to have quadtrees for $S_0, \ldots, S_\ell$. This can be achieved in total linear time as follows: start with the quadtree for $P = S_0$ and in each round i remove the points from $S_{i-1} \setminus S_i$ and prune the tree. □

Theorem 3.3 immediately gives us an improved transdichotomous algorithm for Delaunay triangulations.

THEOREM 3.4. *Suppose our computational model is a word RAM with a constant-time shuffle operation. Let $P \subseteq \{0, \ldots, 2^w - 1\}^d$ be an n-point set. Then* $\mathrm{DT}(P)$ *can be computed in expected time $O(\texttt{sort}(n) + C(P))$, where $C(P)$ denotes the expected structural change of an RIC on P and* $\texttt{sort}(n)$ *denotes the time needed for sorting n numbers.*

PROOF. First use Lemma 3.1 to build a compressed quadtree for P in $O(\texttt{sort}(n))$ time, and then apply Theorem 3.3; see Figure 5. □

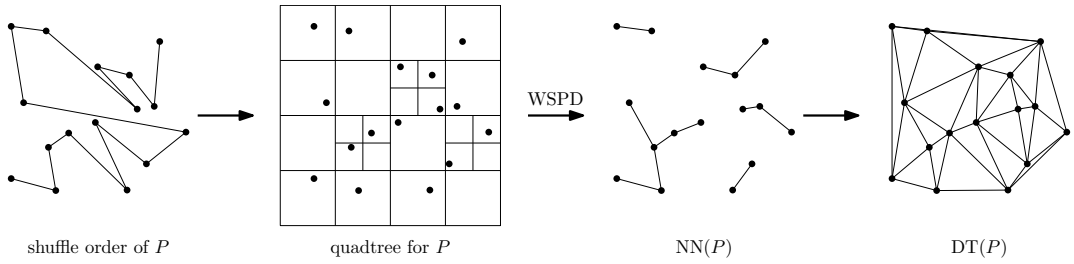

Fig. 5. An illustration of Theorem 3.4: from shuffle sorting to quadtree to well-separated pair decomposition to nearest-neighbor graph to Delaunay triangulation.

Remark 3.5. For planar point sets, $C(P)$ is always linear, and this also often holds in higher dimensions.

[3]The operation `msb` returns the index of the first nonzero bit in a given word, and -1 if the word is 0.

Remark 3.6. In the plane there is another approach to Theorem 3.4, 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. [1994] and Bern et al. [1999] 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 [Chazelle et al. 2002; Chazelle and Mulzer 2009a]. However, the details of the construction of $\mathrm{DT}(P')$ are fairly intricate, and it is not clear how to split $\mathrm{DT}(P')$ efficiently in higher dimensions.

Our reduction has a curious consequence about presorted point sets, since we can find quadtrees for such point sets by an algebraic computation tree [Arora and Barak 2009, Chapter 16.2] of linear depth.

THEOREM 3.7. *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 $O(n + C(P))$, where $C(P)$ denotes the expected structural change of an RIC on P.*

PROOF. Build the quadtree in the standard way [Bern et al. 1994; Callahan and Kosaraju 1995], but with two changes: (i) before splitting a box into 2^d subboxes, shrink it to a smallest enclosing box by using the minimum and maximum values for each coordinate axis (this is more convenient for the algebraic computation tree framework); and (ii) when partitioning the points in each box, use simultaneous exponential searches from both sides. The modification (i) ensures that 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, and it is known that Lemma 3.2 still holds for this modified quadtree (as can also be seen by inspecting the proof presented by Chan [2008, Step 3]). 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 such nodes. Although the algorithm needs only $O(n)$ comparisons, it is not clear how to implement it in time $o(n \log n)$, because while building the tree we must maintain the sorted order of the points in each box according to *every* coordinate axis. □

As mentioned in the introduction, van Emde Boas trees [van Emde Boas et al. 1976; van Emde Boas 1977] give us a way to preprocess a large universe so that its subsets can be sorted more quickly. We will now derive an analogous result for planar Delaunay triangulations.

THEOREM 3.8. *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 $O(u \log u)$ [Bern et al. 1994]. We use T in order to find NNGs quickly. Let $S \subseteq U$ be a subset of size m. The *compressed induced subtree* for S, T_S, is obtained by taking the union of all paths from the root of T to a leaf in S and by compressing all paths with nodes that have only one child. We now show that T_s can be found in time $O(m \log \log u)$.

CLAIM 3.9. *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 compressed induced subtree T_S in time $O(m \log \log u)$.*

PROOF. Build a vEB tree [van Emde Boas et al. 1976; Mehlhorn 1984] for U (according to the order of the leaves in T), and preprocess T into a pointer-based data structure for least-common-ancestor (lca) queries.[4] Recall that lca queries on a pointer machine take $\Theta(\log \log u)$ time per query [Harel and Tarjan 1984; van Leeuwen and Tsakalides 1988]. Furthermore, compute the depth for each node of T and store it with the respective node. These data structures need $O(u \log \log u)$ space. Given S, we use the vEB tree to sort S according to the order of T. Then we use the lca structure to compute T_S as follows: apply `BuildQuadTree` (Algorithm 2) to S according to the sorted order, but with two changes: for the nodes q_k we take the appropriate nodes in T, and instead of `box` we use lca queries. More precisely, we initialize q_0 as the root of T. Whenever we need to compare $\texttt{box}(p_{i-1}, p_i)$ with $q_k.\texttt{box}$, we compute the lca q of p_{i-1} and

[4] Given a rooted tree T and two nodes $u, v \in T$, the *least common ancestor* of u and v is the last node that appears both on the path from the root to u and on the path from the root to v.

p_i to see whether q lies below or above q_k in T (by comparing the depths). Instead of creating a new node q_{k+1} in Step 2b, we just let $q_{k+1} = q$ and set pointers appropriately.

Since there are $O(m)$ queries to the vEB tree, and $O(m)$ queries to the lca structure (as `BuildQuadTree` only executes $O(m)$ steps in total), the whole process takes time $O(m \log\log u)$, as claimed. □

In order to find $\mathrm{NN}(S)$, use Claim 3.9 to compute T_S and then use Theorem 3.3. This takes $O(m \log\log u)$ time, and now the claim follows from Theorem 2.1. □

Remark 3.10. As is well known [de Berg et al. 2008; Boissonnat and Yvinec 1998; Mulmuley 1994; Preparata and Shamos 1985], once we have computed the DT, we can find many other important geometric structures in $O(n)$ time, for example, the Voronoi diagram, the Euclidean minimum spanning tree, or the Gabriel graph. Given the Voronoi diagram of a planar point set P, we can also find the largest empty circle inside $\operatorname{conv} P$ in linear time [Preparata and Shamos 1985].

4. SCAFFOLD TREES

We now extend Theorem 3.8 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*), i.e., no four points of U lie in a common plane and no point $p \in U$ is contained in the convex hull of $U \setminus \{p\}$. We call our data structure the *scaffold tree*. Our description starts with a simple structure that handles convex hull queries in expected 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 $\operatorname{conv} U$ denote the convex hull of U, and let $E[U], F[U]$ be the edges and facets of $\operatorname{conv} U$. 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 $\operatorname{conv} S$ in conflict with p. Furthermore, we call $|B_f|$ and $|D_p|$ the *conflict sizes* of f and p, respectively. We will need a recent result about splitting convex hulls [Chazelle and Mulzer 2009a, 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* $\operatorname{conv} U$*, computes* $\operatorname{conv} P$ *in total expected time $O(u)$.* □

We will also need the following sampling lemma for the recursive construction of the scaffold tree [Chazelle and Mulzer 2009a, Lemma 4.2]. After stating the lemma formally, we will say a bit more about its meaning and how it can be used for the construction of scaffold trees.

LEMMA 4.2. *Let $U \subseteq \mathbb{R}^3$ be a u-point set in gcp, and let $\mu \in (0,1)$ be a constant. There exists a constant α_0 such that the following holds: pick an integer α with $\alpha_0 \le \alpha \le \mu u$. Given* $\operatorname{conv} U$*, in $O(u)$ expected time we can compute subsets $S, R \subseteq U$ and a partition $R_1, \ldots, R_\beta$ of R such that*

(1) $|S| = \alpha$, $|R| = \Omega(u)$, $\max_i |R_i| = O\left(u(\log\alpha)/\alpha\right)$.

(2) For each R_i, there exists a facet $f_i \in F[S]$ such that all points in R_i are in conflict with f_i.

(3) Every point in R conflicts with constantly many facets of $\operatorname{conv} S$.

(4) The conflict sets for points $p \in R_i$, $q \in R_j$, $i \neq j$, are disjoint and no conflict facet of p shares an edge with a conflict facet of q.

We can find $\operatorname{conv} S$, $\operatorname{conv} R_1$, …, $\operatorname{conv} R_\beta$, *and* $\operatorname{conv}(U \setminus (R \cup S))$ *in total expected time $O(u)$.* □

The idea behind Lemma 4.2 is as follows: it is well known that if we take a random subset $S \subseteq U$ of size α, the total expected conflict size for $\operatorname{conv} S$ is $O(u)$, and with high probability the maximum conflict size over all facets in $F[S]$ is $O(u(\log\alpha)/\alpha)$ [Clarkson and Shor 1989; Mulmuley 1994]. In particular, a typical point $p \in U \setminus S$ will conflict with constantly many facets of $\operatorname{conv} S$. Lemma 4.2 says that if we discard a constant fraction of the points (the set $U \setminus (S \cup R)$), not only can we turn this into a worst-case bound, but we can also partition the conflict sets into *independent* parts (the R_i's), which means that no two points in different R_i's share a conflict facet (by Property (4)).

This last property is the key to the scaffold tree. Recall that given an n-subset $P \subseteq U$, our goal is to find $\operatorname{conv} P$. In view of Theorem 4.1, our strategy is to construct the convex hull of a (not too large) superset $P' \supseteq P$, from which we then extract $\operatorname{conv} P$ in expected time $O(|P'|)$. The larger hull $\operatorname{conv} P'$ is obtained through divide and conquer: we use Lemma 4.2 to find an appropriate convex hull $\operatorname{conv} S$, the *scaffold*. As stated in the lemma, $\operatorname{conv} S$ partitions (a constant fraction of) the universe into several parts R_i, and we recurse on the R_i's. This yields subhulls $\operatorname{conv}(P \cap R_i)$ that need to be combined. Here Property (4) is crucial: together with Property (3), it enables us to insert the subhulls $\operatorname{conv}(P \cap R_i)$ into $\operatorname{conv} S$ in total linear time, since we can process each subhull in isolation. Hence, we obtain the hull $\operatorname{conv}(P')$ for $P' = S \cup P$, and we then remove the points in $P' \setminus P$ in expected time $O(|S| + |P|)$. There is a trade-off for the size of S: if the scaffold is too small, we do not make enough progress in the recursion; if it is too large, removing the scaffold takes too long. It turns out that the optimum is achieved for $|S| = 2^{\sqrt{\log u}}$. Finally, note that in Lemma 4.2 we discard a constant fraction of the universe U, so we need to iterate the algorithm several times and then compute the union of the resulting hulls, for which we pay another factor of $\log \log u$. The details follow.

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 expected time $O(n\sqrt{\log u} \log \log u)$. If* $\operatorname{conv} U$ *is known, the preprocessing time is $O(u\sqrt{\log u})$.*

PROOF. We first describe the preprocessing phase. If necessary, we construct $\operatorname{conv} U$ in time $O(u \log u)$. The scaffold tree is computed through the recursive procedure `BuildTree`(U); see Algorithm 3.

Algorithm 3 Building the basic scaffold tree.

`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 $\alpha = 2^{\sqrt{\log u}}$ to obtain subsets $S_i, R_i \subseteq U_i$, as well as a partition $R_i^{(1)}, \ldots, R_i^{(\beta)}$ of R_i, and the hulls $\operatorname{conv} S_i$, $\operatorname{conv} R_i^{(j)}$, $\operatorname{conv}(U_i \setminus (S_i \cup R_i))$ described in the lemma.
  (b) Call `BuildTree`$\left(R_i^{(j)}\right)$ for $j = 1, \ldots, \beta$.
  (c) Let $U_{i+1} = U_i \setminus (S_i \cup R_i)$ and increment i.
(3) Let $\ell = i$ and call `BuildTree`(U_ℓ).

By Property (1) of Lemma 4.2, the sizes of the sets U_i decrease geometrically, so we get $\ell = O(\sqrt{\log u})$ and

$$\sum_{i=1}^{\ell} |S_i| = O\left(2^{\sqrt{\log u}} \sqrt{\log u}\right). \tag{5}$$

The geometric decrease of the U_i's, together with the running time bound from Lemma 4.2, also implies that the total expected time for Step 2a is $O(u)$. Thus, since by Property (1) of Lemma 4.2 the sets for the recursive calls in Steps 2b and 3 have size $O(|U_i|(\log \alpha)/\alpha) = O(u\sqrt{\log u}/2^{\sqrt{\log u}})$, the expected running time $P_2(u)$ of `BuildTree` obeys the recursion

$$P_2(u) = O(u) + \sum_i P_2(m_i),$$

where

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

Thus, the total expected work in each level of the recursion is $O(u)$, and for the number of levels $L(u)$ we get

$$L(u) \leq 1 + L\big(\sigma u\sqrt{\log u}/2^{\sqrt{\log u}}\big),$$

for some constant $\sigma > 0$. To see that $L(u) \le c\sqrt{\log u}$ for some appropriate constant c, we use induction and note that for u sufficiently large we have

$$L(u) \le 1 + c\sqrt{\log u + \log\sigma + (1/2)\log\log u - \sqrt{\log u}} \le 1 + c\sqrt{\log u}\sqrt{1 - 1/(2\sqrt{\log u})} \le c\sqrt{\log u},$$

since for c large enough $c\sqrt{\log u}\sqrt{1 - 1/(2\sqrt{\log u})} \le c\sqrt{\log u} - 1$. Therefore, we get $P_2(u) = O\big(u\sqrt{\log u}\big)$.

Queries are answered by a recursive procedure called `Query`(P); refer to Algorithm 4. Step 1 takes time

Algorithm 4 Querying the simple scaffold tree.

`Query`(P)

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

$O(n \log n) = O(n\sqrt{\log u} + n \log\log u) = O\left(n\sqrt{\log u}\right)$ [de Berg et al. 2008; Boissonnat and Yvinec 1998; Mulmuley 1994; Preparata and Shamos 1985]. 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 [Mehlhorn 1984]), 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 $P_i^{(j)} \ne \emptyset$. We show how to insert $P_i^{(j)}$ into conv S_i in time $O(|P_i^{(j)}|)$. This implies the claim, since Property (4) of Lemma 4.2 combined with simple geometric arguments [Chazelle and Mulzer 2009a, Lemma 4.5] shows that there can be no edge between two points $p \in P_i^{(j_1)}$ and $q \in P_i^{(j_2)}$, for $j_1 \ne j_2$. The only facets in conv S_i that are destroyed are facets in $\Delta = \bigcup_{p \in P_i^{(j)}} D_p$, by definition of D_p. By Properties (2) and (3) of Lemma 4.2, the size of Δ is constant, because all the D_p's have constant size, form connected components in the dual of conv S_i (a 3-regular graph), and have one facet in common. Thus, all we need is to insert the constantly many points in S_i incident to the facets in Δ into conv $P_i^{(j)}$, which takes time $O(|P_i^{(j)}|)$, as claimed. □

By Theorem 4.1, Step 2d needs $O(|P_i| + |S_i|)$ time. Using an algorithm for merging convex hulls [Chazelle 1992], Step 4 can be done in time $O(n \log \ell) = O(n \log\log u)$. Thus, the total expected time for Steps 2 to 4, excluding the recursive calls, is $O(n \log\log u + \sum_{i=1}^{\ell} |S_i|) = O(n \log\log u)$, by (5) and Step 1 of the algorithm, which ensures that P is large enough. Since there are $O(\sqrt{\log u})$ levels in the recursion, 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. As mentioned above, the reason for the $O(\sqrt{\log n})$ factor in the running time of the basic structure lies in a trade-off between the depth of the recursion and the time to remove the scaffold $S \setminus P$. To improve the running time, we increase the size of S in order to make the recursion more shallow. However, now it may happen that S is much larger than P, and hence the application of `SplitHull` might take too long. Recall how we used the scaffold in the basic algorithm: every recursively computed subhull conv$(P \cap R^{(j)})$ is inserted into conv S by connecting it appropriately with the vertices that bound the facets in a conflict set $\Delta^{(j)}$ of constant size. Now, if S is much larger than P, most intersections $P \cap R^{(j)}$ will be empty. Hence, it would suffice to insert the subhulls into a smaller scaffold conv $\widetilde{S}$ with $\widetilde{S} \subseteq S$, so that $\widetilde{S}$ contains only the vertices of the facets in the $\Delta^{(j)}$'s for those $R^{(j)}$'s that meet P. Since the $R^{(j)}$'s

have constant size, $|\widetilde{S}| = O(|P|)$. But how do we construct $\operatorname{conv}\widetilde{S}$? Here is where the bootstrapping comes into play, because finding $\operatorname{conv}\widetilde{S}$ given $\operatorname{conv} S$ is exactly an instance of the problem we are trying to solve in the first place. By choosing the parameters carefully, and by taking the bootstrapping process through several steps in which the size of the scaffold gradually increases, we can improve the multiplicative factor in the running time from $\sqrt{\log u}\log\log u$ to $(\log\log u)^2$. The details are given below. First, however, there is one more subtlety we need to address: for best results, we would not only like $|\widetilde{S}| = O(|P|)$, but $|\widetilde{S}| \le |P|$, so that the problem size does not increase during the bootstrapping process. This is not quite possible, because the conflict facets for a $R^{(j)}$ are bounded by more then one vertex. Nonetheless, the following lemma shows that it suffices to find $\operatorname{conv} S'$ for a subset $S' \subseteq \widetilde{S}$ that contains a random point from each vertex set for a relevant $\Delta^{(j)}$, so that $|S'| \le |P|$.

THEOREM 4.5. *Let $S \subseteq U$ be in gcp, and let $S_1, \ldots, S_k \subseteq S$ such that $|S_i| \le c$, for some constant c and all i, and such that the subgraphs $\operatorname{conv} S|_{S_i}$ are connected and available. Set $m = \sum_{i=1}^{k} |S_i|$. Furthermore, let $S' \subseteq S$ be obtained by choosing one random point from each S_i, uniformly at random, and suppose $\operatorname{conv} S'$ is available, and that we have a van Emde Boas structure for the neighbors of each vertex in $\operatorname{conv} S$. Then we can find $\operatorname{conv}\left(\bigcup_{i=1}^{k} S_i\right)$ in expected time $O(m\log\log u)$.*

Note that the S_i's are not necessarily disjoint.

PROOF. The idea is to insert the remaining points one by one. As shown below, the total structural change for these insertions turns out to be linear. To determine the locations where the new points need to be inserted in the current hull, we use the information provided by the subgraphs $\operatorname{conv} S|_{S_i}$. Details follow.

Let Q be a queue which we initialize with the points from S'. While Q is not empty, let p be the next point in Q. The point p may be contained in several sets S_i. We insert the neighbors for p in each subgraph $\operatorname{conv} S|_{S_i}$ that contains p into the vEB structure for p, thus sorting all the neighbors according to clockwise order. Let $q_1, q_2, \ldots, q_\beta$ denote the neighbors in order. We insert the q_j's one by one into the current hull. For q_1, we inspect all facets incident to p in the current hull until we find one that conflicts with q_1, and then insert q_1 in the standard way. By assumption, pq_1 is an edge of the current hull, and we inspect the facets incident to p in the current hull in clockwise order, starting from those incident to pq_1, until we find one that is in conflict with q_2. Then we insert q_2 and proceed in the same way. This ensures that any facet incident to p is inspected at most constantly often. Finally, we append all the q_j that have not been encountered yet to the queue Q.

The total time taken is proportional to $m\log\log u$ for the sorting and the traversal of the queue, plus the number of facets of the convex hull that were created (and possibly destroyed) during the construction.

Let f be a facet with conflict set B_f (with respect to $\bigcup_{i=1}^{k} S_i$). The facet f is created only if $S' \cap B_f = \emptyset$. The probability of this event is at most

$$\prod_{i=1}^{k}\left(1 - \frac{|S_i \cap B_f|}{|S_i|}\right) \le \exp\left(-\sum_{i=1}^{k}\frac{|S_i \cap B_f|}{c}\right) \le \exp(-|B_f|/c).$$

Now, since it is well known that there are at most $O(ms^2)$ facets with conflict size s [Clarkson and Shor 1989], the expected number of created facets is $O\left(m \cdot \sum_{s=0}^{\infty} s^2/e^{s/c}\right) = O(m)$, and the result follows. □

THEOREM 4.6. *Let $k \ge 2$ be an integer and $U \subseteq \mathbb{R}^3$ 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 $P_k(u)$ and query time $Q_k(n,u)$ such that*

$$P_k(u) \le \beta u k l_k \text{ and } Q_k(n,u) \le \beta n k l_k \log\log u,$$

then there is a data structure $D_{k+1}(U)$ with preprocessing time $P_{k+1}(u)$ and query time $Q_{k+1}(n,u)$ such that

$$P_{k+1}(u) \le \beta u(k+1)l_{k+1} \text{ and } Q_{k+1}(n,u) \le \beta n(k+1)l_{k+1}\log\log u.$$

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 $k \le 0.7\log\log u - 1$, because 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 $\alpha = 2^{(\log u)^{k/(k+1)}}$; and (iii) for each sample S_i we compute vEB trees for the neighbors of each vertex in $\operatorname{conv} S_i$, and also a data structure $D_k(S_i)$ for convex hull queries, which exists by assumption; for details see Algorithm 5.

Algorithm 5 Bootstrapping the scaffold tree.

$\texttt{BuildTree}_{k+1}(U)$

(1) If $|U| = O(1)$, store U and return, otherwise, 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, with $\alpha = 2^{(\log u)^{k/(k+1)}}$. This yields subsets $S_i, R_i \subseteq U_i$, a partition $R_i^{(1)}, \ldots, R_i^{(\beta)}$ of R_i, and the convex hulls $\operatorname{conv} S_i$, $\operatorname{conv} R_i^{(j)}$, and $\operatorname{conv}(U_i \setminus (S_i \cup R_i))$ with the properties of Lemma 4.2.

  (b) Execute $\texttt{BuildTree}_{k+1}\left(R_i^{(j)}\right)$ for $j = 1, \ldots, \beta$, compute a data structure $D_k(S_i)$ for S_i as well as van Emde Boas structures for the neighbors of each vertex in $\operatorname{conv} S_i$.

  (c) Let $U_{i+1} = U_i \setminus (S_i \cup R_i)$ and increment i.

(3) Let $\ell = i$ and call $\texttt{BuildTree}_{k+1}(U_\ell)$.

Since by Property (1) of Lemma 4.2 the sizes of the U_i's decrease geometrically, we have $\ell = O\left(l_{k+1}^k\right)$ and the total expected time for Step 2a is $O(u)$, by the running time guarantee from Lemma 4.2. Since $2 \le k \le 0.7 \log\log u - 1$, we have

$$2^{0.7} \le l_k, l_{k+1} \le \sqrt{\log u},$$

and therefore the total time to construct the data structures $D_k(S_i)$ in Step 2b is at most (for some large enough constant c)

$$c\left(l_{k+1}^k\right) P_k\left(2^{l_{k+1}^k}\right) \le c\left(l_{k+1}^k\right) \beta k l_{k+1} 2^{l_{k+1}^k} \le c\beta(\log u \log\log u) \cdot 2^{(\log u)/l_{k+1}} \le c\beta(\log u \log\log u) \cdot u^{1/2^{0.7}} \le u,$$

for u large enough. Similarly, the total time for the construction of the vEB trees in Step 2b can be bounded by $O\left(l_{k+1}^k 2^{l_{k+1}^k} \log\log u\right) \le u$, again for u large enough. Since by Property (1) of Lemma 4.2 the sets $R_i^{(j)}$ all have size at most $O\left(u l_{k+1}^k / 2^{l_{k+1}^k}\right)$, the number of levels $L_{k+1}(u)$ has

$$L_{k+1}(u) \le 1 + L_{k+1}\left(u\sigma(\log u)/2^{l_{k+1}^k}\right),$$

for some constant σ (note that the term l_{k+1}^k depends on u, since $l_{k+1} = (\log u)^{1/(k+1)}$). To prove that $L_{k+1}(u) \le c(k+1)l_{k+1}$, for some constant $c > 0$, we use induction to write

$$L_{k+1}(u) \le 1 + c(k+1)\left(\log u + \log\sigma + \log\log u - l_{k+1}^k\right)^{1/(k+1)}.$$

As $k \ge 2$, we have $l_{k+1}^k = (\log u)^{k/(k+1)} \ge \sqrt{\log u}$, so $\log\sigma + \log\log u - l_{k+1}^k < -0.9 l_{k+1}^k$, for u large enough, and

$$L_{k+1}(u) \le 1 + c(k+1)l_{k+1}\left(1 - \frac{0.9}{l_{k+1}}\right)^{1/(k+1)}.$$

Now, since

$$\left(1 - \frac{0.9}{l_{k+1}}\right)^{1/(k+1)} \le 1 - \frac{0.9}{(k+1)l_{k+1}}, \tag{6}$$

by the standard inequality $(1+x)^r \le 1 + rx$ for $x > -1$ and $0 < r < 1$, we get

$$L_{k+1}(u) \le 1 + c(k+1)l_{k+1} - 0.9c \le c(k+1)l_{k+1},$$

for c large enough. The total expected work at each level is $O(u)$, so the total preprocessing time is $P_{k+1}(u) \le \beta u(k+1)l_{k+1}$, for β large enough.

Algorithm 6 Querying the bootstrapped scaffold tree.

$\texttt{Query}_{k+1}(P)$

(1) For $i = 1, \ldots, \ell - 1$
   (a) Let $P_i = P \cap R_i$ and determine the intersections $P_i^{(j)}$ of P_i with the sets $R_i^{(j)}$.
   (b) For each nonempty $P_i^{(j)}$, if $|P_i^{(j)}| \leq \beta k l_{k+1}$, compute the hull $\operatorname{conv} P_i^{(j)}$ directly, otherwise call $\texttt{Query}_{k+1}(P_i^{(j)})$.
   (c) For each nonempty $P_i^{(j)}$, determine the set $S_i^{(j)}$ of all points adjacent to a conflict facet of $P_i^{(j)}$ in $F[S_i]$.
   (d) Let S_i' be a set that contains one random point from each $S_i^{(j)}$. Use $D_k(S_i)$ to find $\operatorname{conv} S_i'$.
   (e) Let $\widetilde{S}_i = \bigcup_j S_i^{(j)}$. Use Theorem 4.5 to compute $\operatorname{conv} \widetilde{S}_i$ and then Claim 4.4 to find $\operatorname{conv}(P_i \cup \widetilde{S}_i)$.
   (f) Use `SplitHull` to extract $\operatorname{conv} P_i$ from $\operatorname{conv}(P_i \cup \widetilde{S}_i)$.
(2) Recursively compute $\operatorname{conv} P_\ell$, where $P_\ell = U_\ell \cap P$.
(3) Compute $\operatorname{conv} P$ as the union of $\operatorname{conv} P_1, \ldots, \operatorname{conv} P_\ell$.

Queries are answered by $\texttt{Query}_{k+1}$, as shown in Algorithm 6. In the following, let c denote a large enough constant. As before, Step 1a takes time

$$T_{1a} \leq cn, \tag{7}$$

using 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

$$T_{1b} \leq cn_1 \log\log u + cn_1 \log \beta + Q_{k+1}\left(n_2, cul_{k+1}^k / 2^{l_{k+1}^k}\right), \tag{8}$$

assuming that Q_{k+1} is linear in the first and monotonic in the second component (which holds by induction on the second component). By Properties (2) and (3) of Lemma 4.2, the sets $S_i^{(j)}$ are subsets of constant-sized sets that can be precomputed, so Step 1c takes time

$$T_{1c} \leq cn. \tag{9}$$

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

$$T_{1d} \leq Q_k\left(n_1 + \frac{n_2}{\beta k l_{k+1}}, 2^{l_{k+1}^k}\right) \leq \beta k \left(n_1 + \frac{n_2}{\beta k l_{k+1}}\right) l_{k+1} \log\log u = (\beta n_1 k l_{k+1} + n_2) \log\log u, \tag{10}$$

because $\left(\log(2^{l_{k+1}^k})\right)^{1/k} = (\log u)^{k/(k(k+1))} = l_{k+1}$. In Step 1e, we use Theorem 4.5 to find $\operatorname{conv} \widetilde{S}_i$ in expected time $O(|\widetilde{S}_i| \log\log u)$, from which we can compute $\operatorname{conv}(P_i \cup \widetilde{S}_i)$ in time $O(|P_i|)$, as in the proof of Claim 4.4. By Theorem 4.1, extracting $\operatorname{conv} P_i$ needs $O(|P_i| + |\widetilde{S}_i|)$ expected time, so Steps 1e and 1f take total time

$$T_{1e,1f} = O\left(n + \sum_i |\widetilde{S}_i| \log\log u\right) \leq cn \log\log u, \tag{11}$$

as $|\widetilde{S}_i| = O(|P_i|)$ for all i. Step 2 is already accounted for by T_{1b}. Finally, Step 3 requires

$$T_3 \leq cn \log\log u \tag{12}$$

time. By summing (7,8,9,10,11,12) we get the following recurrence for $Q_{k+1}(n, u)$:

$$Q_{k+1}(n, u) \leq 5cn \log\log u + cn_1 \log \beta + \beta n_1 k l_{k+1} \log\log u + Q_{k+1}\left(n_2, cu \frac{l_{k+1}^k}{2^{l_{k+1}^k}}\right). \tag{13}$$

By induction, we get

$$Q_{k+1}\left(n_2, cu\frac{l_{k+1}^k}{2^{l_{k+1}^k}}\right) \leq \beta n_2(k+1)\left(\log u + \log c + \frac{k}{k+1}\log\log u - l_{k+1}^k\right)^{1/(k+1)}\log\log u.$$

As $l_{k+1}^k \geq \sqrt{\log u}$, we have $\log c + \frac{k}{k+1}\log\log u - l_{k+1}^k \leq -0.9 l_{k+1}^k$, for u large enough. Using (6), it follows that

$$\begin{aligned} Q_{k+1}\left(n_2, cu\frac{l_{k+1}^k}{2^{l_{k+1}^k}}\right) &\leq \beta n_2(k+1)l_{k+1}\left(1 - \frac{0.9}{l_{k+1}}\right)^{1/(k+1)}\log\log u \\ &\leq \beta n_2(k+1)\left(l_{k+1} - \frac{0.9}{k+1}\right)\log\log u \\ &= \beta n_2(k+1)l_{k+1}\log\log u - 0.9\,\beta n_2\log\log u. \end{aligned}$$

Plugging this bound into (13), we get

$$Q_{k+1}(n,u) \leq 5cn\log\log u + cn_1\log\beta + \beta n_1 k l_{k+1}\log\log u + \beta n_2(k+1)l_{k+1}\log\log u - 0.9\,\beta n_2\log\log u.$$

Now for β large enough, we get

$$\begin{aligned} 5cn\log\log u + cn_1\log\beta &\leq (\beta/2)n_2\log\log u + (\beta/2)n_1\log\log u + (\beta/2)n_1 \\ &\leq (\beta/2)n_2\log\log u + \beta n_1\log\log u, \end{aligned}$$

so

$$\begin{aligned} Q_{k+1}(n,u) \leq (\beta/2)n_2\log\log u + \beta n_1\log\log u + \beta n_1 k l_{k+1}\log\log u + \\ \beta n_2(k+1)l_{k+1}\log\log u - 0.9\,\beta n_2\log\log u \leq \beta n(k+1)l_{k+1}\log\log u, \end{aligned}$$

as claimed. □

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 $O(u\log\log u)$, and if $\operatorname{conv} U$ is available, the preprocessing time reduces to $O(u\log\log u)$.*

PROOF. For $k = \frac{1}{2}\log\log u$, we have $(\log u)^{1/k} = O(1)$, and the result follows from Theorem 4.3 and a repeated application of Theorem 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 $O(n(\log\log n)^2)$.*

PROOF. Given $\operatorname{conv} P$, build the structure from Corollary 4.7 in $O(n\log\log n)$ time. Then perform a query for each color class. Since the color classes are disjoint, the total time is $O(n(\log\log n)^2)$. □

ACKNOWLEDGMENTS

We would like to thank Bernard Chazelle, David Eppstein, Jeff Erickson, and Mikkel Thorup for stimulating and insightful discussions. We would also like to thank an anonymous referee for numerous insightful comments that helped improve the presentation of the paper, and for suggesting a simplification for the proof of Claim 3.9.

REFERENCES

AGGARWAL, A. 1988. Lecture notes in computational geometry. Tech. Rep. 3, MIT Research Seminar Series MIT/LCS/RSS.

ALBERS, S. AND HAGERUP, T. 1997. Improved parallel integer sorting without concurrent writing. *Inform. and Comput. 136,* 1, 25–51.

AMENTA, N., ATTALI, D., AND DEVILLERS, O. 2007. Complexity of Delaunay triangulation for points on lower-dimensional polyhedra. In *Proc. 18th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*. ACM-SIAM, New Orleans, LA, USA, 1106–1113.

AMENTA, N., CHOI, S., AND ROTE, G. 2003. Incremental constructions con BRIO. In *Proc. 19th Annu. ACM Sympos. Comput. Geom. (SoCG)*. ACM, San Diego, CA, USA, 211–219.

Amir, A., Efrat, A., Indyk, P., and Samet, H. 2001. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica 30,* 2, 164–187.

Andersson, A., Hagerup, T., Nilsson, S., and Raman, R. 1998. Sorting in linear time? *J. Comput. System Sci. 57,* 1, 74–93.

Arora, S. and Barak, B. 2009. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, UK.

Attali, D. and Boissonnat, J.-D. 2004. A linear bound on the complexity of the Delaunay triangulation of points on polyhedral surfaces. *Discrete Comput. Geom. 31,* 3, 369–384.

Batcher, K. E. 1968. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conferences*. ACM, Atlantic City, NJ, USA, 307–314.

Ben-Or, M. 1983. Lower bounds for algebraic computation trees. In *Proc. 16th Annu. ACM Sympos. Theory Comput. (STOC)*. ACM, Boston, MA, USA, 80–86.

Bentley, J. L., Weide, B. W., and Yao, A. C. 1980. Optimal expected-time algorithms for closest-point problems. *ACM Trans. Math. Softw. 6*, 563–580.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. 2008. *Computational Geometry: Algorithms and Applications*, Third ed. Springer-Verlag, Berlin.

de Berg, M., van Kreveld, M., and Snoeyink, J. 1995. Two- and three-dimensional point location in rectangular subdivisions. *J. Algorithms 18,* 2, 256–277.

Bern, M., Eppstein, D., and Gilbert, J. 1994. Provably good mesh generation. *J. Comput. System Sci. 48,* 3, 384–409.

Bern, M., Eppstein, D., and Teng, S.-H. 1999. Parallel construction of quadtrees and quality triangulations. *Internat. J. Comput. Geom. Appl. 9,* 6, 517–532.

Boissonnat, J.-D. and Yvinec, M. 1998. *Algorithmic Geometry*. Cambridge University Press, Cambridge, UK.

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

Buchin, K. 2008. Delaunay triangulations in linear time? (part I). `arXiv:0812.0387`.

Buchin, K. 2009. Constructing Delaunay triangulations along space-filling curves. In *Proc. 17th Annu. European Sympos. Algorithms (ESA)*. Springer-Verlag, Copenhagen, Denmark, 119–130.

Callahan, P. B. and Kosaraju, S. R. 1995. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM 42,* 1, 67–90.

Chan, T. M. 2002. Closest-point problems simplified on the RAM. In *Proc. 13th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*. ACM-SIAM, San Francisco, CA, USA, 472–473.

Chan, T. M. 2008. Well-separated pair decomposition in linear time? *Inform. Process. Lett. 107,* 5, 138–141.

Chan, T. M. and Pătraşcu, M. 2009. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput. 39,* 2, 703–729.

Chan, T. M. and Pătraşcu, M. 2007. Voronoi diagrams in $n2^{O(\sqrt{\lg \lg n})}$ time. In *Proc. 39th Annu. ACM Sympos. Theory Comput. (STOC)*. ACM, San Diego, CA, USA, 31–39.

Chazelle, B. 1992. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM J. Comput. 21,* 4, 671–696.

Chazelle, B., Devillers, O., Hurtado, F., Mora, M., Sacristán, V., and Teillaud, M. 2002. Splitting a Delaunay triangulation in linear time. *Algorithmica 34,* 1, 39–46.

Chazelle, B. and Mulzer, W. 2009a. Computing hereditary convex structures. In *Proc. 25th Annu. ACM Sympos. Comput. Geom. (SoCG)*. ACM, Aarhus, Denmark, 61–70.

Chazelle, B. and Mulzer, W. 2009b. Markov incremental constructions. *Discrete Comput. Geom. 42,* 3, 399–420.

Chew, L. P. and Fortune, S. 1997. Sorting helps for Voronoi diagrams. *Algorithmica 18,* 2, 217–228.

Clarkson, K. L. 1983. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*. IEEE, Tucson, AZ, USA, 226–232.

Clarkson, K. L. and Seshadhri, C. 2008. Self-improving algorithms for Delaunay triangulations. In *Proc. 24th Annu. ACM Sympos. Comput. Geom. (SoCG)*. ACM, College Park, MD, USA, 148–155.

Clarkson, K. L. and Shor, P. W. 1989. Applications of random sampling in computational geometry. II. *Discrete Comput. Geom. 4,* 5, 387–421.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2009. *Introduction to Algorithms*, Third ed. MIT Press, Cambridge, MA, USA.

Djidjev, H. N. and Lingas, A. 1995. On computing Voronoi diagrams for sorted point sets. *Internat. J. Comput. Geom. Appl. 5,* 3, 327–337.

van Emde Boas, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett. 6,* 3, 80–82.

van Emde Boas, P., Kaas, R., and Zijlstra, E. 1976. Design and implementation of an efficient priority queue. *Math. Systems Theory 10,* 2, 99–127.

Fredman, M. L. 1976. How good is the information theory bound in sorting? *Theoret. Comput. Sci. 1,* 4, 355–361.

Fredman, M. L. and Willard, D. E. 1993. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci. 47,* 3, 424–436.

Gabow, H. N., Bentley, J. L., and Tarjan, R. E. 1984. Scaling and related techniques for geometry problems. In *Proc. 16th Annu. ACM Sympos. Theory Comput. (STOC)*. ACM, Washington, DC, USA, 135–143.

Han, Y. 2004. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms 50,* 1, 96–105.

Han, Y. and Thorup, M. 2002. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. 43rd Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*. IEEE, Vancouver, BC, Canada, 135–144.

Harel, D. and Tarjan, R. E. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput. 13,* 2, 338–355.

Iacono, J. and Langerman, S. 2000. Dynamic point location in fat hyperrectangles with integer coordinates. In *Proc. 12th Canad. Conf. Comput. Geom. (CCCG)*. CCCG, Fredericton, NB, Canada, 181–186.

Isenburg, M., Liu, Y., Shewchuk, J. R., and Snoeyink, J. 2006. Streaming computation of Delaunay triangulations. *ACM Trans. Graph. (Proc. ACM SIGGRAPH) 25,* 3, 1049–1056.

Karlsson, R. G. 1985. Algorithms in a Restricted Universe. Ph.D. thesis, University of Waterloo.

Karlsson, R. G. and Overmars, M. H. 1988. Scanline algorithms on a grid. *BIT 28,* 2, 227–241.

Kirkpatrick, D. and Reisch, S. 1984. Upper bounds for sorting integers on random access machines. *Theoret. Comput. Sci. 28,* 3, 263–276.

van Kreveld, M. J., Löffler, M., and Mitchell, J. S. B. 2008. Preprocessing imprecise points and splitting triangulations. In *Proc. 19th Annu. Internat. Sympos. Algorithms Comput. (ISAAC)*. Springer-Verlag, Gold Coast, Australia, 544–555.

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

Liu, Y. and Snoeyink, J. 2005. A comparison of five implementations of 3d Delaunay tesselation. In *Combinatorial and Computational Geometry*, J. E. Goodman, J. Pach, and E. Welzl, Eds. MSRI Publications, vol. 52. Cambridge University Press, Cambridge, UK, 439–458.

Löffler, M. and Mulzer, W. 2011. Triangulating the square: quadtrees and Delaunay triangulations are equivalent. Proc. 22nd Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA). To appear.

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

Miller, G. L., Teng, S.-H., Thurston, W., and Vavasis, S. A. 1997. Separators for sphere-packings and nearest neighbor graphs. *J. ACM 44,* 1, 1–29.

Morton, G. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Tech. rep., IBM Ltd., Ottawa, Canada.

Motwani, R. and Raghavan, P. 1995. *Randomized algorithms*. Cambridge University Press, Cambridge, UK.

Mulmuley, K. 1994. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, Upper Saddle River, NJ, USA.

Ohya, T., Iri, M., and Murota, K. 1984a. A fast Voronoi-diagram algorithm with quaternary tree bucketing. *Inform. Process. Lett. 18,* 4, 227–231.

Ohya, T., Iri, M., and Murota, K. 1984b. Improvements of the incremental method for the Voronoi diagram with a comparison of various algorithms. *J. Operations Res. Soc. Japan 27*, 306–337.

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

Preparata, F. P. and Shamos, M. I. 1985. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, Berlin, Germany.

Raman, R. 1996. Priority queues: Small, monotone and trans-dichotomous. In *Proc. 4th Annu. European Sympos. Algorithms (ESA)*. Springer-Verlag, Barcelona, Spain, 121–137.

Schönhage, A. 1979. On the power of random access machines. In *Proc. 6th Internat. Colloq. Automata Lang. Program. (ICALP)*. Springer-Verlag, Graz, Austria, 520–529.

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

Su, P. and Drysdale, R. 1997. A comparison of sequential Delaunay triangulation algorithms. *Comput. Geom. Theory Appl. 7*, 361–386.

Thorup, M. 1998. Faster deterministic sorting and priority queues in linear space. In *Proc. 9th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*. ACM-SIAM, San Francisco, CA, 550–555.

Willard, D. E. 2000. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput. 29,* 3, 1030–1049.

Zhou, S. and Jones, C. B. 2005. HCPO: an efficient insertion order for incremental Delaunay triangulation. *Inform. Process. Lett. 93,* 1, 37–42.

A. SHUFFLE SORTING ON A WORD RAM

We show how to sort a point set $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^d$ according to the shuffle order in expected time $O(n \log\log n)$ on a standard word RAM, without using the shuffle operation. For this, we adapt a classic sorting algorithm by Andersson et al. [1998]. This algorithm consists of two parts: (i) *range reduction*, which reduces the number of bits needed to represent the coordinates; and (ii) *packed sorting*, which packs many

points into one word to speed up sorting. We describe how to adapt these steps and how to obtain the sorting algorithm from them. Let $w \geq \log n$ be the word size and v the number of bits required to represent the coordinates of the p_i's (i.e., the coordinates of the p_i's are in the range $\{0, \ldots, 2^v - 1\}$). We assume that w and v are known.

A.1 Packed sorting for large words

The following theorem is a simple extension of a result by Albers and Hagerup [1997] and shows that we can sort in linear time if many points fit into one word.

THEOREM A.1. *Suppose that $v \leq \lfloor w/(d \log n \log\log n) \rfloor - 1$. Then P can be sorted according to the shuffle order in $O(n)$ time.*

PROOF. By assumption, we can store $2k = \lfloor w/(dv+1) \rfloor \geq \lfloor \log n \log\log n \rfloor$ *fields* in one word,[5] where each field consists of a point preceded by a *test bit*. Given a word a, we let $a[1], \ldots, a[2k]$ denote the $2k$ fields in a, and $a[i].\texttt{test}, a[i].\texttt{data}$ represent the test bit and data stored in field i, respectively. The main ingredient is a procedure for merging two sorted words in logarithmic time [Albers and Hagerup 1997, Section 3].

CLAIM A.2. *Given two words, each containing a sorted sequence of k points, we can compute a word storing the sequence obtained by merging the two input sequences in time $O(\log k)$.*

PROOF. The proof relies on a parallel implementation of a bitonic sorting network by Batcher [1968]. For this, we need to solve the following problem: given two words a and b, each containing a sequence of k points, compute in constant time a word z such that[6] $z[i].\texttt{test} = [a[i].\texttt{data} <_\sigma b[i].\texttt{data}]$ for $i = 1, \ldots, k$, i.e., the test bit of $z[i]$ indicates whether the point in $a[i]$ precedes the point in $b[i]$ in the shuffle order.[7]

Algorithm 7 Comparing many points simultaneously.

`BatchCompare`

(1) Create d copies $a_1, \ldots, a_d$ of a and d copies $b_1, \ldots, b_d$ of b. Shift and mask the words a_j, b_j so that $a_j[i].\texttt{data}$ and $b_j[i].\texttt{data}$ contain only the j-th coordinates of $a[i].\texttt{data}$ and $b[i].\texttt{data}$, for $i = 1, \ldots, k$.

(2) Create words `curMax` and `maxIdx` such that $\texttt{curMax} = a_1 \oplus b_1$ and $\texttt{maxIdx} = (1, \ldots, 1)$, i.e., the word with a 1 in the `data` item of all its fields. (Recall that $\oplus$ denotes bitwise `xor`.)

(3) For $j = 2, \ldots, d$:

  (a) Create a word $c_j = a_j \oplus b_j$. Set the test bits in c_j such that $c_j[i].\texttt{test} = [\texttt{curMax}[i].\texttt{data} \leq c_j[i].\texttt{data}]$ and compute a mask $M_{(4)\text{a}}$ for the fields i with $c_j[i].\texttt{test} = 1$.

  (b) Create a word $d_j = \texttt{curMax} \oplus a_j \oplus b_j$. Set the test bits in d_j so that $d_j[i].\texttt{test} = [\texttt{curMax}[i].\texttt{data} < d_j[i].\texttt{data}]$ and compute a mask $M_{(4)\text{b}}$ for the fields i with $d_j[i].t = 1$.

  (c) Let $M_{(4)} = M_{(4)\text{a}}$ `and` $M_{(4)\text{b}}$. Set

$$\texttt{curMax} = (\texttt{curMax and } \overline{M_{(4)}}) \texttt{ or } ((a_j \oplus b_j) \texttt{ and } M_{(4)}),$$

  where $\overline{M_{(4)}}$ is the bitwise negation of $M_{(4)}$. Furthermore, set

$$\texttt{maxIdx} = (\texttt{maxIdx and } \overline{M_{(4)}}) \texttt{ or } ((j, \ldots, j) \texttt{ and } M_{(4)}).$$

(4) Create a word z with $z[i].\texttt{test} = [a_{\texttt{maxIdx}[i]}[i] < b_{\texttt{maxIdx}[i]}[i]]$.

We solve this problem with an algorithm `BatchCompare` that is based on a technique by Chan [2002]. Recall that we can test whether $a[i].\texttt{data} <_\sigma b[i].\texttt{data}$ by first determining the smallest coordinate j of $a[i].\texttt{data}$ and $b[i].\texttt{data}$ such that the index of the highest bit in which the j-th coordinates of $a[i].\texttt{data}$ and $b[i].\texttt{data}$ differ is maximum. Knowing j, we can test whether $a[i].\texttt{data} <_\sigma b[i].\texttt{data}$ by comparing the j-th coordinates of $a[i].\texttt{data}$ and $b[i].\texttt{data}$. This idea is implemented by `BatchCompare` using (4). Refer to

[5] We assume that the number of fields is even, without loss of generality.
[6] We use Iverson's notation: $[X] = 1$ if X is true and $[X] = 0$, otherwise.
[7] We denote the shuffle order by $<_\sigma$.

Algorithm 7. Steps 1 and 2 create appropriate words that store only the individual coordinates and initialize the current maximum to the first coordinate. The search for the actual maximum takes place in the loop of Step 3. To find the index of the highest bit in which $a[i]$ and $b[i]$ differ, we need to consider $a[i] \oplus b[i]$, and we use (4) to determine whether this index exceeds the current maximum. The equation (4) is a conjunction of two terms. These two terms are implemented in Steps 3a and 3b. This yields a mask $M_{(4)}$ that contains 1's for all fields that pass the in test (4) (i.e., those fields for which the current maximum needs to be replaced by j). The maxima are updated in Step 3c. The final comparison happens in Step 4.

`BatchCompare` runs in constant time, assuming that the constants $(1, \ldots, 1)$, ..., $(d, \ldots, d)$ have been precomputed, which can be done in $O(\log k)$ time. In particular, note that Step 4 takes constant time since there are only constantly many possible values for `maxIdx`$[i]$. □

Given Claim A.2, the theorem now follows by an application of merge sort, see Albers and Hagerup [1997] for details. □

A.2 Range reduction

In order to pack several points into a word, we need to adapt a range reduction technique due to Kirkpatrick and Reisch [1984].

THEOREM A.3. *With expected $O(n)$ time overhead, the problem of shuffle sorting n points with v-bit coordinates can be reduced to the problem of sorting n points with $v/2$-bit coordinates. The space needed for the reduction is $O(n)$.*

PROOF. The proof is verbatim as in the paper by Kirkpatrick and Reisch [1984, Section 4]: we bucket the points according to the upper $v/2$ bits of their coordinates. Using universal hashing, this can be done in $O(n)$ expected time with $O(n)$ space. From each nonempty bucket b, we select the maximum element it contains, m_b. We truncate the coordinates of each m_b to the upper $v/2$ bits, and store a flag `isMaximum` in the satellite data for m_b. The coordinates of the remaining points are truncated to the lower $v/2$ bits, and the number of their corresponding bucket is stored in the satellite data. After the resulting point set has been sorted, we use the satellite data to establish (i) the ordering of the buckets, by using the sorted maxima, and (ii) the ordering within each bucket. The crucial fact needed for correctness is that 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 $v/2$ bits. □

A.3 Putting it together

Following Andersson et al. [1998], we combine the algorithms in Sections A.1 and A.2 to obtain a simple randomized $O(n \log\log n)$ sorting algorithm.

THEOREM A.4. *Given a set P of n points with v-bit integer coordinates, we can sort P in expected time $O(n \log\log n)$ with $O(n)$ space.*

PROOF. Iterate Theorem A.3 $O(\log\log n)$ times until $O(\log n \log\log n)$ points fit into one word. Then apply Theorem A.1. The total space for the range reduction is $O(n)$ because in each step the number of bits for the satellite data is halved. □

Received Month Year; revised Month Year; accepted Month Year