Delaunay triangulations on the word RAM: Towards a practical worst-case optimal algorithm

Okke Schrijvers
Computer Science Dept.
Stanford University
Palo Alto, California
okkes@cs.stanford.edu

Frits van Bommel
Dept. Mathematics and Computer Science
Technische Universiteit Eindhoven
The Netherlands

Kevin Buchin
Dept. Mathematics and Computer Science
Technische Universiteit Eindhoven
The Netherlands
k.a.buchin@tue.nl

***Abstract*—The Delaunay triangulation of n points in the plane can be constructed in $o(n \log n)$ time when the coordinates of the points are integers from a restricted range. However, algorithms that are known to achieve such running times had not been implemented so far. We explore ways to obtain a practical algorithm for Delaunay triangulations in the plane that runs in linear time for small integers. For this, we first implement and evaluate two variants of BrioDC, an algorithm that is known to achieve this bound. We implement the first $O(n)$-time algorithm for constructing Delaunay triangulations and found that our implementations are practical. While we do not improve upon fast existing algorithms (with non-optimal worst-case running time) for realistic data sets, our BrioDC implementations do give us insight into the optimal time needed for point location. Secondly, we implement and evaluate variants of BRIO, an algorithm which has an $O(n \log n)$ worst-case running time on small integers but runs faster for many distributions. Our variants aim to avoid bad worst-case behavior, which is due to high point location time. Our BrioDC implementation shows that point location time can be reduced by 25% and our squarified space-filling curve orders show the first improvement by reducing this by 3%.**

I. Introduction

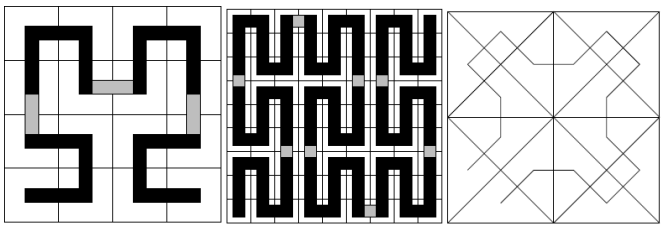
The Delaunay triangulation (DT) of a point set P in the plane is a triangulation of P such that no point $p \in P$ lies inside the circumcircle of any triangle. In general, constructing the Delaunay triangulation of n points in the plane takes $\Omega(n \log n)$ time. This bound holds if we assume that the coordinates of the points are arbitrary real numbers. However, in the word RAM model of computation, that is, if we assume that the coordinates of the points are integers from a restricted range $[0, U)$ this bound no longer holds. Nonetheless, for a long time no algorithms that beat this bound were known. The breakthrough came in 2006, when Chan and Pătraşcu [9] presented an algorithm running in $O(n \log n / \log \log n)$ time. In a follow-up paper [10] they improved this bound to $n2^{O(\sqrt{\log \log n})}$. The best asymptotic running time to hope for is the time for sorting integers in the range $[0, U)$. A randomized and a deterministic algorithm achieving this in a suitable model were given by Buchin and Mulzer [6] and Löffler and Mulzer [19], respectively.

For small integers the latter two algorithms run in linear time, since we can sort integers with $U = n^{O(1)}$ in this time using radix sort. Integer sorting algorithms like radix sort are not only fast in theory, but also run fast in experiments (see for example [17]). In contrast, $o(n \log n)$-time algorithms for Delaunay triangulations have been only of theoretical interest so far and none of them had been implemented. The goal of our work is to explore ways to obtain a practical algorithm for Delaunay triangulations in the plane that runs in linear time for small integers. Our approach to this is two-fold. First, we implement and evaluate fast variants BrioDC [6], which has the same asymptotic running time as sorting. Secondly, we implement and evaluate variants of incremental constructions *con BRIO* [2], where BRIO stands for *biased randomized insertion order*.

Incremental constructions con BRIO (when used without point-location data structure) also seem suited for points with small integer coordinates, because their worst-case expected running time is in $O(n \log U)$ [5][1] (in contrast to the $O(n \frac{\log U}{\log n})$ running time of BrioDC). The variants we have implemented aim to avoid typical reasons for bad worst-case behavior. For real-valued coordinates the worst-case running time of incremental constructions con BRIO is quadratic but can be reduced to $O(n \log n)$ by the use of a point location data structure [2][2]. There are various implementations of variants of this algorithm [2], [4], [11], [16], [18], [27]. Most of these variants do not use an additional point location data structure and most sort the points of a round along a space-filling curve (SFC) like the Hilbert or Peano curve (see Figure 1).

[1] In [5] this bound is formulated in terms of the spread of the point set.

[2] They prove a corresponding result in 3D, but their analysis can be extended to two-dimensional Delaunay triangulations (and other configuration spaces) as shown in [4].



(a) Hilbert (b) Peano (c) Sierpiński

Fig. 1. Traditional space-filling curves.

In experiments these variants mostly seem to run in linear time, but unfortunately there are point sets for which the $O(n \log U)$-bound is tight [5]. Therefore, if we want an algorithm with a better worst-case performance, we need to choose the insertion order differently.

One weakness of orders based on space-filling curves seems to be that the construction process does not adapt to the point distribution. In contrast, the CGAL Hilbert curve order [11] does[3]. However, this order is likely to introduce large jumps, i.e., large distances between consecutive points in the order. We propose several new orders that overcome this problem and still adapt well to the point distribution. For comparison we also implemented various traditional space-filling curve orders.

The algorithm by Buchin and Mulzer [6] propose an extension of BRIOs that uses nearest neighbor graphs (NNGs). We implement their algorithm. A large part of the running time is spent on the worst-case optimal nearest-neighbor graph construction, and we therefore implement another variant of this algorithm which uses several steps that are non-optimal but simpler. In our experiments the running time of both variants is similar.

II. Algorithms

We implemented both the BRIO and BrioDC algorithms[4]. For each we give a short overview and discuss variations and implementation details.

A. BRIO

The difference between a regular *random incremental construction* (RIC) algorithm and BRIO [2], is that with BRIO the points are inserted in $\log_2 n$ rounds. First, points are added to the final round with independent probability $1/2$. Then the remaining points are added with probability $1/2$ to the second-to-last round and this process continues until we reach the first round and all remaining points are added. Within a round the insertion order can be chosen freely, and often a space-filling curve order is used. By sorting points in this way, the next point to be inserted is likely to be close to the previous point.

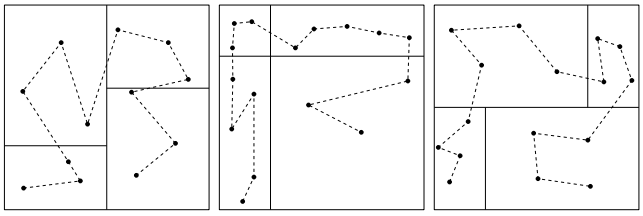
1) Existing Space-Filling Curves: There is a plethora of space-filling curves available in the literature; we have implemented the following to use with BRIO.

a) Hilbert: Perhaps the most widely known curve is the Hilbert curve [15] (Figure 1a). It subdivides the square into four smaller squares and visits them in cyclic order. In the recursive step, the cyclic order may start at a different square and go in a different direction.

b) Peano: The Peano curve [21] subdivides the square into a 3×3 grid and roughly follows the pattern of the letter N (Figure 1b). Each subregion follows a similar order, except possibly reflected along the Y-axis.

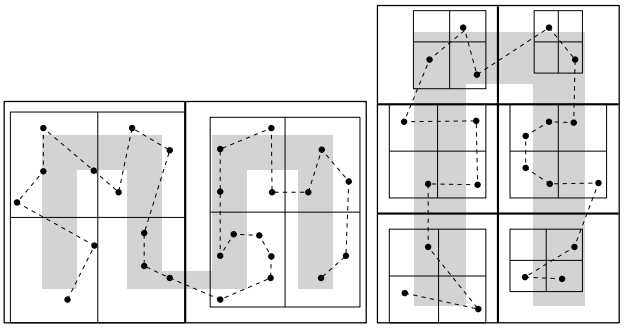
[3]CGAL now also provides a regular Hilbert curve order.

[4]Source code available from http://www.win.tue.nl/~kbuchin/proj/wramdt/.



(a) CGAL (b) Adaptive (c) HilbertYX

Fig. 2. Existing variants of the Hilbert curve.



(a) Wide squarified (b) Tall squarified

Fig. 3. Squarified Hilbert curve.

c) Sierpiński: Unlike curves that are based on subdividing squares into smaller squares, the Sierpiński curve [25] subdivides triangles into smaller triangles (Figure 1c). It starts by subdividing the initial square into two triangles along a diagonal, and from then on only works with triangles.

d) CGAL Hilbert: CGAL [11] provides an space-filling curve that is similar to the Hilbert order, but instead of dividing the space into equally sized subspaces, it divides the point set into equally sized subsets. It first creates a vertical split on the horizontal median, and then for each half a horizontal split on their vertical medians. The quadrants are handled in Hilbert order, see Figure 2a. If the two vertical medians differ substantially, the jump from the upper left to upper right quadrant can be large.

2) New space-filling curves: We propose several new space-filling curves that aim at providing a good mapping between 1-dimensional and d-dimensional space, i.e., no jumps should occur and the summed distance between consecutive points in the order should be small. The running time is related to the depth of recursion [5]. To reduce the depth, our methods aim at providing balanced splits and using tighter bounding boxes. A more detailed description of the new space-filling curves is given in [26].

a) Adaptive Hilbert order: As a compromise between the original and CGAL Hilbert orders, the entire point set is split by its horizontal and vertical median, see Figure 2b. This should distribute the points over all quadrants better than the original Hilbert order, and remove the jump that was seen in CGAL Hilbert.

b) Hilbert YX: The reason the jump occurs in CGAL Hilbert is because the point set is split along the x-axis first,

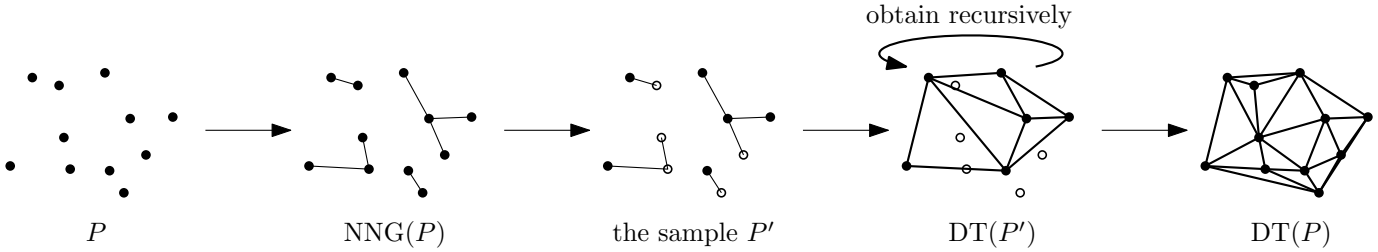


Fig. 4. A schematic representation of the BrioDC algorithm (adapted from [6]). Our implementations differ in the way they compute NNG.

and then along the y-axis, creating a discontinuity at the top. By first splitting on the y-axis median and then on the two x-axis medians, no jump occurs when going from one quadrant to another, see Figure 2c.

c) Squarified Hilbert and Peano orders: If the smallest enclosing bounding box of a point set has large aspect ratio, fitting an space-filling curve will either leave part of the domain unused, or stretch one axis. We prevent this from happening by splitting the point set into subsets that have a well-fitting square bounding box, and joining the space-filling curves over these point sets. Depending on whether the bounding box is wide or tall (relative to the orientation of the space-filling curve) this is achieved by placing either multiple curves next to each other, as in Figure 3a, or two columns of multiple curves, as in Figure 3b. The bounding box is recomputed for each subproblem, so the individual subproblems may be split in a similar way. Squarifying can also be combined with the adaptive orders.

3) Implementation: BRIO was implemented in C++ and uses CGAL 3.6.1 to generate the Delaunay triangulation after the rounds have been determined. We implemented our orders as sorting operations. For the CGAL Hilbert order we use `hilbert_sort_2`. We use CGAL to obtain a direct comparison between our orders and the CGAL Hilbert order. The code of CGAL was only slightly modified to gather metrics.

B. BrioDC

BRIO determines the rounds in which the points are inserted using independent probabilities. On the other hand, BrioDC uses a series of nearest-neighbor graphs to determine the rounds. As with BRIO we construct the rounds backwards, starting with the final round. Let $\text{NNG}_{\leq r}$ be the nearest-neighbor graph of all points that are added in rounds 1 to r. We then determine which points will be inserted in round r as follows. First, we identify all the connected components in the nearest-neighbor graph. For each connected component we make sure that at least one point is inserted before round r by picking either the first or second point at random. The other point will be added in round r. For all subsequent points we add them to round r with independent probability 1/2. We know that for each connected component in $\text{NNG}_{\leq r}$, at least one point is inserted in an earlier round. This process is repeated for each round until the point set is of constant size, for which we generate the Delaunay triangulation directly. This process is given as a recursive procedure in Algorithm 1 and as a schematic representation in Figure 4.

BRIODC(P)
1 If $|P| = O(1)$, then compute DT(P) directly and return
2 Compute NNG(P), the nearest-neighbor graph for P
3 Let $P' \subset P$ be a random sample such that
  (i) P' meets every connected component of NNG(P)
  (ii) $Pr[p \in P'] = 1/2$ for all $p \in P$
4 Call BRIODC(P') to compute DT(P).
5 Compute DT(P) by inserting the points of $P \backslash P'$ into DT(P), using NNG(P) as a guide.

Algorithm 1: The BrioDC algorithm as given by Buchin and Mulzer [6].

For each round r we insert the points using $\text{NNG}_{\leq r}$ as a guide. We traverse $\text{NNG}_{\leq r}$ breadth first, starting at all points p_i that were inserted in an earlier round. If we follow an edge from p to q where p is already in the Delaunay triangulation, and q is not, we insert q using p as a guiding vertex. We do this by walking from point p to the triangle in which q lies and start updating the Delaunay triangulation from there. Alternatively, since $\text{NNG}_{\leq r}$ is a subset of $\text{DT}_{\leq r}$, we could remove all triangles intersecting the line segment $\overline{pq}$ and start fixing the Delaunay triangulation from there.

1) Variations: The running time of BRIODC is determined by the time spent in the generation of the nearest-neighbor graph [6]. Therefore, we look into two different algorithms that compute this.

a) Linear-time algorithm: To find the nearest-neighbor graph of a point set, we use a series of intermediate data structures as illustrated in Figure 5. From the point set we generate a *compressed quadtree*, from which we compose a *well-separated pair decomposition (WSPD)* that is used to compute the nearest-neighbor graph. Obviously, for a linear-time algorithm, each step can take at most linear time.

For the compressed quadtree we use the approach of Chan [8], that builds the compressed quadtree bottom-up. We first sort all points along a z-order space-filling curve [20]. We only need information on the most significant bit in which the z-index of two consecutive points differ, to determine the size and position of the quadtree box. Constructing the compressed quadtree in this way takes linear time, plus the time needed

for sorting the points. For integer points this also takes linear time using radix sort. We implement radix sort using lookup tables for the coordinates, as described by Schrijvers [23].

We compute the well-separated pair decomposition from the compressed quadtree as described by Callahan and Koseraju [7]. This algorithm takes linear time in the number of elements in the quadtree, and as we use a compressed quadtree, this is linear in the number of points. We actually do not need all the well-separated pair decomposition pairs, as we will discuss in the following paragraph. To reduce the required storage, we only save pairs where at least one element is a singleton. This way we reduce the required storage of the well-separated pair decomposition by a factor of approximately 30.

Finally, we follow the approach of Callahan and Koseraju [7] to compute the nearest-neighbor graph from the well-separated pair decomposition. This approach is based on the observation that the nearest neighbor of a point p must appear in the point set B for a pair (p, B) in the well-separated pair decomposition. Additionally, when looking at B, there can only be a constant k (based on the bounding circle of B) of points p_i, for which their potential nearest neighbor is in B. If there are too many points around B, they will be closer to each other than to points in B, hence no points from B can be a nearest-neighbor. We can maintain this information efficiently by using the quadtree for hierarchical information. Since the number of potential nearest neighbors that we need to maintain is bounded by the constant k and the size of the quadtree and well-separated pair decomposition is linear, this can be maintained in linear time. The final step is to determine which of the k potential nearest neighbors is closest, which can also be done in total linear time. So for integer points, we can compute the nearest-neighbor graph and identify all connected components in linear time.

b) $O(n \log U)$-time algorithm: While all the algorithms in the previous subsection take linear time, some parts involve a large constant. We also implemented the following asymptotically inferior algorithms with lower constants.

For the generation of the compressed quadtree, we can simply first build the regular quadtree, using a standard incremental algorithm. The running time per point is dependent on the maximum depth of the quadtree, which is $\log U$. There are examples where every point takes as much time, so size and running time of the quadtree is $O(n \log U)$. Computing the compressed quadtree takes $O(n \log U)$ after which its size is linear.

For computing the nearest-neighbor graph from the well-separated pair decomposition, we can also check for every pair (p, B), which point in B is closest to p. This takes $O(n(\log n + \log U))$ [14]. Since $\log n$ is bounded by $\log U$, this can be simplified to $O(n \log U)$.

2) Implementation: We have implemented the BrioDC algorithm in C++. For the base case of the Delaunay triangulation and for the incremental insertion of points into the triangulation we use the *Triangle* library[5] [24] version 1.6. Triangle gives the option to supply a guiding vertex when inserting a new point. We use the guiding point q from the nearest-neighbor graph and walk to the triangle containing the new point p. While it may be more efficient to first remove any triangles intersecting the line segment $\overline{pq}$, we used walking so we do not have to modify Triangle. Other than the modifications to gather the measurements, we have made no changes to the library.

[5]http://www.cs.cmu.edu/~quake/triangle.html

III. Experimental Setup

In this section we discuss the experimental setup. We start by discussing the different data distributions we have used and we conclude this section with a description of the metrics that we have collected. All code was written in C++ and compiled using GCC with the -O3 optimization flag. The experiments were performed on a 64-bit 16 core Intel Xeon L5520 server running Linux (2.6.35) operating system with 11.7 gigabytes of RAM.

A. Distributions

When generating point sets on an integer domain, there is a possibility of two points being mapped to the same coordinates. When this occurs we record the event and generate a new point according to the chosen distribution. For each dataset we generate point sets of size $n = 2^{10}$ to 2^{22} in powers of 2. The domain is $[0, U) \times [0, U)$ where $U = 2^8 \cdot n$. For each distribution and n we run 10 tests on different datasets and report the average of the results.

We tested the following point distributions (Figure 6).

a) Uniform and Checkers: We have tested uniformly distributed point sets in a square domain (Figure 6a). A variation on this is the checkers distribution where 1/8 of the points are distributed uniformly on white squares and 7/8 of the points are distributed uniformly on black squares (Figure 6b).

b) Normal: We take both the x and y coordinate independently from a normal distribution (Figure 6c). For this we use the Box-Muller transform as described in "Numerical Recipes in C" [22]. We rescale the point set by 50 standard deviations, which did not result in points that were discarded due to rescaling.

c) Kuzmin: Blelloch et al. [3] describe the Kuzmin distribution as a radially symmetric distribution with high density center. In Figure 6d we show the center of the domain. For each point the radius r is determined by picking a uniform random variable $x \in [0, 1)$ and solving $x = 1 - \frac{1}{\sqrt{1-r^2}}$; the angle is taken uniformly at random in $[0, 2\pi)$ radians. We rescale the dataset by $U/2500$. This means that on average we discard 5-10 points because they fall outside the domain. The number of points discarded because they are mapped to the same integer coordinate scales linearly with n: approximately 1 in every 2^{12} points is rejected.

d) Line: Blelloch et al. [3] define a line singularity to give an example of a distribution that has a convergence area. Most of the points appear on the same line segment, with a few points appearing to the right of it (Figure 6e). It is obtained by

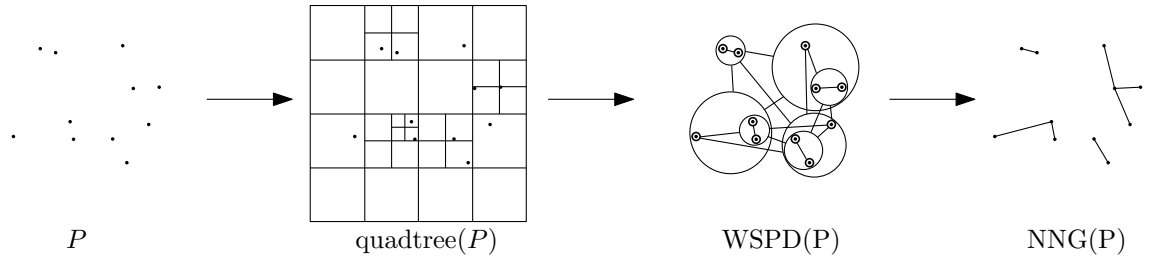
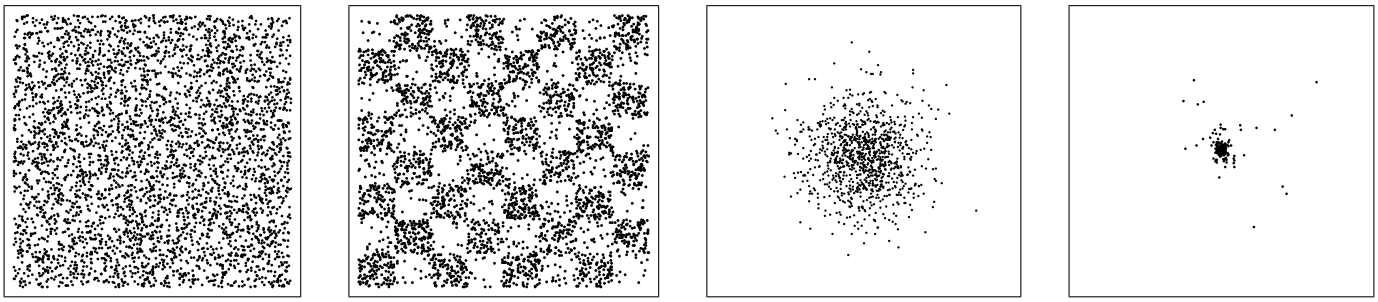


Fig. 5. A schematic representation of the data structures used to compute NNG: a (compressed) quadtree and a well-separated pair decomposition. Every well-separated pair is represented by a pair of circles connected by an edge.



(a) uniform (b) checkers (c) normal (d) Kuzmin

(e) line (f) worst-case (g) lidar

Fig. 6. Our different point set distributions. For the normal and Kuzmin distributions we show the center of the domain, otherwise all points would appear as a single point. The lidar data shown is a subsample.

taking two independent random variables u and v in the range $[0, 1]$ and applying the transformation $(x, y) = \left(\frac{b}{u-bu+b}, v\right)$, with $b = 0.001$.

e) Worst-case Hilbert: Buchin [5] describes a worst-case input for BRIO using a Hilbert curve. The basic building block is a set of points on a line of slope -1 with exponentially decreasing distances. For small integer coordinates several copies of this building block are used to achieve the $\Omega(n \log U)$ worst case. For each building block, we use $U = 2^{25}$ and 32×32 copies. A 4×4 example is given in Figure 6f.

f) LIDAR: Delaunay triangulations are often used for terrain modelling. We have used the LIDAR data set 'Yosmemite National Park, CA: Rockfall Studies (CA10_Zimmer)'[6] projected onto the xy-plane (z denotes height) and restricted to the range from (272751.755, 4179980.625) to (273651.446, 4180814.474). The data set contains 4,060,441 points after removing duplicate (x, y) values.

B. Measurements

The largest factor in the practicality of an algorithm is its *running time*. The non-linear worst-case running time in RICs and BRIO is due to the time needed for point location. Since all incremental algorithms that we compare use *walking* for point location, i.e., they find the new point by traversing the triangulation, we measure the point location cost by counting the number of simplices walked.

Measurements that we took but do not report on further are *tour length* and the number of *in-circle* tests. The tour length is the length of the tour through the points in the order given by the BRIO. For BrioDC we took the sum of lengths of the relevant edges of nearest-neighbor graphs. However, tour length in our experiments was proportional to the number of

[6]LiDAR data acquisition and processing completed by the National Center for Airborne Laser Mapping (NCALM – http:\www.ncalm.org). NCALM funding provided by NSF's Division of Earth Sciences, Instrumentation and Facilities Program. EAR-1043051. Data set provided by the OpenTopography Facility with support from the National Science Foundation under NSF Award Numbers 0930731 & 0930643.

simplices walked. The number of in-circle tests corresponds to the cost of updating the triangulation. This number varied only little over the various insertion orders.

IV. Results

We first compare BRIOs with each other. Since they basically fall into two groups in terms of running time, we take a few representative BRIOs and then only compare those in terms of their running time with BrioDC and other algorithms.

A. Comparison between BRIOs

We first compare the various insertion orders for incremental constructions con BRIO using the average simplices walked per point and the average running time per point in microseconds for $n = 2^{22}$. For both types of measurements the squarified orders and the original space-filling curve orders consistently perform better than the other orders. Only for the worst-case point set the original space-filling curve orders perform considerably worse. For the random distributions the performance of the BRIOs varied only little except for the line distribution which we will discuss separately. For the other random distributions we use the normal distribution as a representative, and all numbers in the text refer to this distribution if not stated otherwise.

In terms of the average number of simplices walked, the Squarified Hilbert order performs best. More specifically, the squarified orders (Squarified Hilbert: 3.54, Squarified Peano: 3.65) enforce a walk with fewer simplices than the corresponding original orders (Hilbert: 3.62, Peano: 3.66). Sierpiński (3.56) performs worse than Squarified Hilbert. The squarified adaptive orders walk slightly more simplices (Squarified Adaptive Hilbert: 3.66, Squarified Adaptive Peano: 3.75). The remaining orders walk about 10-20% more simplices (Adaptive Hilbert: 4.16, Adaptive Peano: 4.30, HilbertYX: 4.39, CGAL Hilbert: 4.26). Table I shows how the average number of simplices walked changes for $n = 2^{15}, \ldots, 2^{22}$. While the original space-filling curve orders and squarified (and adaptive squarified) orders show little dependency on the distribution, the number of simplices walked degenerates for the line distribution for the adaptive orders (Adaptive Hilbert: 24.46, Adaptive Peano: 31.59, HilbertYX: 26.74, CGAL Hilbert: 25.54). Interestingly, for adaptive and CGAL orders this number increases with n, while for the squarified and original space-filling curve orders it does not. As comparison, the table also shows the number of triangles walked in the BrioDC algorithm, i.e., starting at the nearest neighbor. This is approximately 25% less than the squarified and original orders.

In terms of the running time, the squarified and original Hilbert orders perform best. In general, the squarified orders (Squarified Hilbert: 1.10 μs, Squarified Peano: 1.11 μs) are slightly slower than the original orders (Hilbert: 1.06 μs, Peano: 1.10 μs, Sierpiński: 1.07 μs). This can be mostly attributed to the higher construction time, but also the average number of conflicts increases slightly (e.g., Hilbert: 4.15 and squarified Hilbert: 4.16). Most remaining orders are slower by 6-26% (Adaptive Hilbert: 1.27 μs, Adaptive Peano: 1.33 μs, HilbertYX: 1.21 μs, CGAL Hilbert: 1.18 μs, Squarified Adaptive Hilbert: 1.29 μs, Squarified Adaptive Peano: 1.34 μs), and this factor is about 75-230% for the line distribution. For the worst-case point set the Squarified Hilbert (1.20 μs) and the Squarified Peano (1.27 μs) are about 30% faster than the corresponding original orders (Hilbert: 1.53 μs, Peano: 1.52 μs). But the Sierpiński order remains fast (1.26 μs).

TABLE I
Average number of simplices walked per algorithm for normally distributed point sets of different size for CGAL Hilbert (CGALHil), Adaptive Hilbert (AdHil), Adaptive Squarified Hilbert (AdSqHil), original Hilbert (Hilbert), Squarified Hilbert (SqHil) and BrioDC.

Input Size	CGALHil	AdHil	AdSqHil	Hilbert	SqHil	BrioDC
2^{15}	4.18	4.08	3.68	3.63	3.55	2.73
2^{16}	4.19	4.10	3.67	3.63	3.55	2.73
2^{17}	4.20	4.10	3.67	3.62	3.54	2.73
2^{18}	4.22	4.12	3.67	3.62	3.54	2.73
2^{19}	4.23	4.13	3.67	3.62	3.54	2.73
2^{20}	4.24	4.14	3.66	3.62	3.54	2.72
2^{21}	4.25	4.15	3.66	3.62	3.54	2.72
2^{22}	4.26	4.16	3.66	3.62	3.54	2.72

The fact that we were not able to construct a large point set for the worst-case Hilbert distribution leads us to an interesting observation. The worst-case construction of the Hilbert curve requires a point set for which point-to-point distances exponentially vary (in the size of the point set). The minimal and maximal distance between any two distinct points is limited by the precision of the number representation. In our case, since we use points from the 32-bit integer domain, we can only use about 30 points. In fact, for our particular construction we take points from $2^{25} \times 2^{25}$, leading to 28 points per construction. This severely reduces the size of the worst-case point set, thereby preventing the $\Omega(n \log U)$ to be a problem in a practical sense. For points with floating point precision, we can construct a point set of roughly the size of the difference in maximum and minimum exponent. According to the IEEE 754 standard [1], for single precision this leads to around 256 points while for double precision we get around 2048. For point with floating point coordinates the running time is in $\Omega(n^2)$ [5], but $n \leq 2048$. This leaves the interesting question if we can find a distribution for which a construction in Hilbert order yields a $\Omega(n^2)$ algorithm, while not bounding the number of points in the distribution.

B. Comparison with existing algorithms

We now compare our BrioDC algorithms with existing ones. The BRIO orders fall into two categories and the orders within the same category have similar running time. Therefore, from the original and squarified orders we will only show Squarified Hilbert and from the adaptive orders we will only show CGAL Hilbert. Figure 7 shows the running time per point of the algorithms for the line distribution on a log-log-scale and Figure 8 shows a lin-lin scale. Table II shows the running times for the different distributions of size 2^{20}, 2^{21} and 2^{22}.

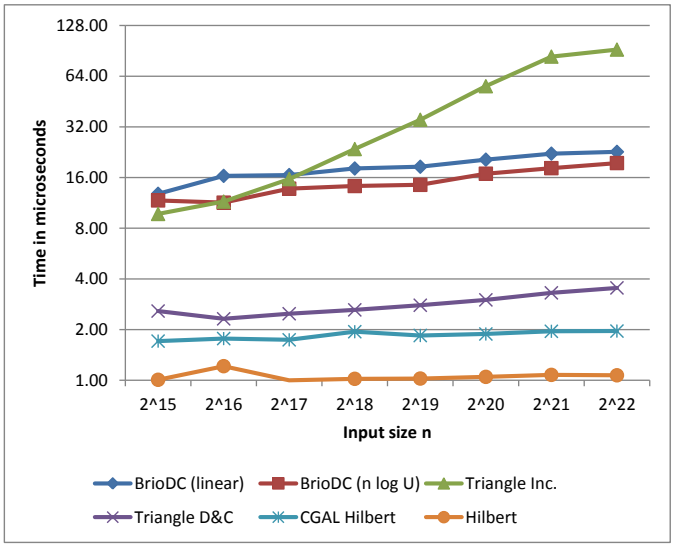


Fig. 7. The running time per point for the different algorithms.

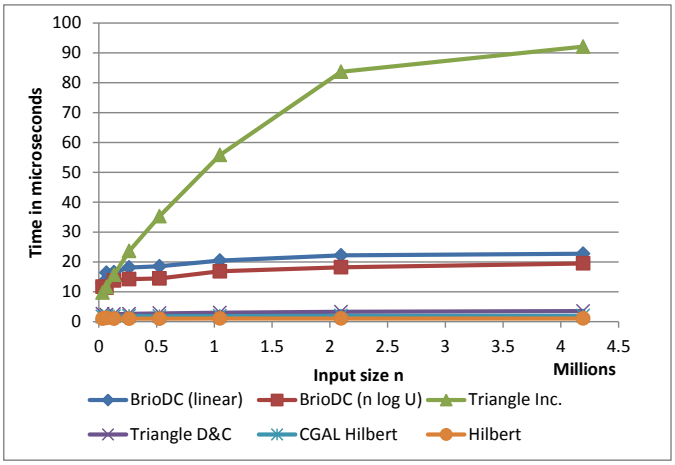


Fig. 8. The running time per point (on a linear scale) for the different algorithms.

Furthermore, it shows the running times for the two additional point sets: worst-case and lidar. For all algorithms based on con BRIO (including BrioDC) we would expect a constant, or nearly constant running time per point. This seems indeed to be the case, although the graph increases very slightly with the input size. The near-linear $O(n \log U)$ time implementation of BrioDC is faster than the linear-time implementation, but is also growing faster. For 2^{19} and more points, BrioDC is faster than the incremental construction algorithm of Triangle. For 2^{22} points BrioDC is only a factor 5.5-7 slower than the divide-and-conquer algorithm of Triangle, and this factor decreases with increasing number of points. The fastest algorithms in our experiments are the algorithms using a BRIO. The Hilbert order, the squarified Hilbert order and BrioDC show little dependency on the input distribution. Triangle's incremental algorithm, HilbertYX and CGAL Hilbert show a large dependency, and the divide-and-conquer algorithm considerably

TABLE II
THE RUNNING TIME PER POINT IN MICROSECONDS FOR BRIODC LINEAR (DCLIN) AND $O(n \log U)$ (DCLOG), TRIANGLE INCREMENTAL (TR. INC.) AND DIVIDE-AND-CONQUER (TR. D&C), CGAL HILBERT (CGALHIL), SQUARIFIED HILBERT (SQHIL) AND HILBERT.

Input Size	DCLin	DCLog	Tr. Inc.	Tr. D&C	CGALHil	SqHil	Hilbert
uniform							
2^{20}	22.26	18.20	32.54	2.22	1.14	1.08	1.05
2^{21}	22.18	18.35	40.52	2.39	1.17	1.09	1.06
2^{22}	23.01	19.79	45.56	2.61	1.15	1.08	1.05
normal							
2^{20}	20.99	18.46	30.64	2.23	1.13	1.06	1.03
2^{21}	22.14	19.33	39.74	2.43	1.17	1.08	1.05
2^{22}	22.77	20.15	55.23	2.73	1.18	1.10	1.06
Kuzmin							
2^{20}	21.89	20.40	29.93	2.38	1.22	1.14	1.10
2^{21}	22.89	21.14	44.70	2.62	1.25	1.16	1.12
2^{22}	23.08	20.64	61.03	2.87	1.24	1.15	1.11
checkers							
2^{20}	20.47	16.42	26.34	2.21	1.17	1.11	1.08
2^{21}	23.83	19.46	48.16	2.58	1.23	1.15	1.12
2^{22}	24.00	21.33	54.18	2.79	1.22	1.14	1.10
line							
2^{20}	20.44	16.87	55.81	3.00	1.88	1.07	1.05
2^{21}	22.20	18.20	83.66	3.31	1.95	1.11	1.08
2^{22}	22.74	19.51	92.10	3.54	1.96	1.10	1.07
worst-case							
28,674	8.77	11.01	6.21	1.71	1.22	1.20	1.53
lidar							
4,060,440	19.73	14.20	6.18	2.05	1.28	1.18	1.17

slows down for the line distribution. For the worst-case input the $O(n \log U)$-BrioDC is slower than the linear one. This is not surprising since the O($\log U$)-factor is related to the depth of the quadtree used in the construction. This depth is large for this point set. All other algorithms perform as expected for the size of the point set, except for the previously discussed space-filling curve orders. For the LIDAR data, $O(n \log U)$-BrioDC and Triangle's incremental construction perform notably better than for other distributions. This is likely due to locality in the data set, which results from the LIDAR data capturing process.

To gain more insight in how to speed up BrioDC, we analyze its running time in terms of the individual steps (Figure 9). The cost is split into constructing the quadtree, the well-separated pair decomposition (from the quadtree) and the nearest-neighbor graph (from the well-separated pair decomposition). Everything else including the sampling is reported as Rest. For the contribution of the well-separated pair decomposition and the nearest-neighbor graph computation this time indeed seems to be close to constant. For the quadtree and remaining calculations this time increases but this growth stalls and seems bounded by a constant. We see that all three steps considerably contribute to the running time.

V. DISCUSSION

We set out to find a practical worst-case optimal Delaunay triangulation algorithm for points with coordinates from a polynomially-bounded integer range. For this we have implemented the BrioDC algorithm, which had only been of theoretical interest so far. We have found that using the nearest-neighbor graphs of each round in BrioDC, the number of

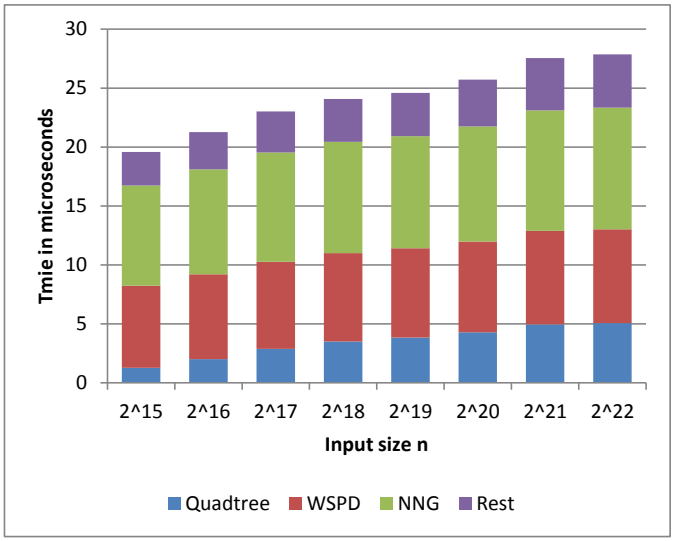


Fig. 9. The running time per point for different parts of BrioDC.

simplices walked is reduced by more than 25% compared to the fastest known space-filling curves we have tested. This gives us an important target, since it implies that the running time for point location can only be improved by 25%, hence any additional computation time that is required to reduce the point location time must only incur a very small overhead.

We compared the running time of our algorithm with Triangle's divide-and-conquer approach which is one of the fastest $O(n \log n)$-time algorithms for constructing Delaunay triangulations. Other commonly used $O(n \log n)$-time algorithms come within a factor of 1.5 to 10 of this running time [12]. With a factor of about 5.5-7 for point sets of size 2^{22} BrioDC is competitive with these algorithms, and this factor will decrease further with increasing input sizes. We have determined that most time is spend on computing the nearest-neighbor graphs through the intermediate data structures of the compressed quadtree and well-separated pair decomposition (so the total of "Quadtree", "WSPD" and "NNG" in Figure 9). It is an open problem whether or not it is possible to skip one of the steps in computing the nearest-neighbor graph. We know the data structures are equivalent, but can we e.g. compute the nearest-neighbor graph directly from the Quadtree, skipping the well-separated pair decomposition? Any progress in this area would substantially improve our running time. From a practical point of view it is also interesting to improve this by using a parallel algorithm that could run on a coprocessor, e.g. a GPU approach like the one by Garcia et al. [13] which is publicly available[7].

We have also compared our implementation against several BRIO orders. One of the reasons that BrioDC uses the nearest-neighbor graph, is because it is expected to reduce the total simplices walked when compared to a space-filling curve. Our experiments show that this is indeed the case. Since the number of simplices walked is that between a point and it's nearest neighbor, it is likely that one cannot do better in a randomized setting. Therefore, we now have a concrete goal for future BRIO approaches to try and reduce this gap, while limiting other computations to not exceed the 25% mark.

Finally, we found that while there exists a $\Omega(n^2)$ bound on Delaunay triangulation constructions using a Hilbert order, in practice the construction imposes an upper bound of $n = O(\log \Phi)$, where Φ is the spread of the underlying number format, i.e. the largest representable number divided by the smallest representable number. It remains an open question if a construction exists that does not bound n in this way.

We have given the first implementation of a $O(n)$-time algorithm for computing the Delaunay triangulation of points with bounded integer coordinates. While currently BrioDC is still outperformed by $O(n \log n)$-time algorithms, this would change with a faster nearest-neighbor graph algorithm. While resorting to a parallel algorithm would achieve this, computing the nearest-neighbor graph fast sequentially (without using the Delaunay triangulation) remains an open problem.

[7]http://www.i3s.unice.fr/~creative/KNN/

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 29.

[2] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. 19th Annu. ACM Sympos. Comput. Geom. (SoCG)*, pages 211–219, San Diego, CA, USA, 2003. ACM.

[3] G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24:243–269, 1999.

[4] K. Buchin. *Organizing Point Sets: Space-Filling Curves, Delaunay Tessellations of Random Point Sets, and Flow Complexes*. PhD thesis, Free University Berlin, 2007. http://www.diss.fu-berlin.de/diss/receive/FUDISS_thesis_000000003494.

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

[6] K. Buchin and W. Mulzer. Delaunay triangulations in $O(sort(n))$ time and more. *J. ACM*, 58:6:1–6:27, April 2011.

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

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

[9] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. C.*, 39(2):703–729, 2009.

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

[11] C. Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 2007.

[12] O. Devillers. The Delaunay hierarchy. *Int. J. Found. Comp. Sc.*, 13(2):163–180, 2002.

[13] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *CVPR W. on Comp. Vis. on GPU*, Anchorage, Alaska, USA, June 2008.

[14] S. Har-peled. *Geometric Approximation Algorithms*. Mathematical Surveys and Monographs. American Mathematical Society, 2011.

[15] D. Hilbert. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[16] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, July 2006.

[17] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *Proc. 15th Internat. Sympos. String Processing and Information Retrieval*, SPIRE '08, pages 3–14, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[19] M. Löffler and W. Mulzer. Triangulating the square: quadtrees and Delaunay triangulations are equivalent. *Proc. 22nd Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1759–1777, 2011.

[20] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

[21] G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890.

[22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[23] O. Schrijvers. Insertions and deletions in delaunay triangulations using guided point location. Master's thesis, TU Eindhoven, Netherlands, 2012.

[24] J. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In M. Lin and D. Manocha, editors, *Applied Computational Geometry Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer Berlin / Heidelberg, 1996.

[25] W. Sierpiński. Sur une nouvelle courbe continue qui remplit toute une aire plane. *Bull. de l'Acad. des Sciences de Cracovie Série A*, pages pp. 463–478, 1912.

[26] F. van Bommel. Biased randomized insertion orders. Master's thesis, TU Eindhoven, Netherlands, 2011.

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