EINDHOVEN UNIVERSITY OF TECHNOLOGY

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

Citation for published version (APA):
Schrijvers, O. J., Bommel, van, F., & Buchin, K. (2012). Delaunay triangulations on the word RAM: towards a practical worst-case optimal algorithm. In *Abstracts 28th European Workshop on Computational Geometry (EuroCG 2012, Assisi, Italy, March 19-21, 2012)* (pp. 13-16). Università degli Studi di Perugia.

Document status and date:
Published: 01/01/2012

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 01. Dec. 2020

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

Okke Schrijvers Frits van Bommel Kevin Buchin

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 variants of an algorithm, BrioDC, that is known to achieve this bound. We find that our implementations of these algorithms are competitive with fast existing algorithms. Secondly, we implement and evaluate variants of an algorithm, BRIO, that runs fast in experiments. Our variants aim to avoid bad worst-case behavior and our squarified orders indeed provide faster point location.

1 Introduction

In general, constructing the Delaunay triangulation (DT) of n points in the plane takes $\Omega(n \log n)$ time. However, if 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 [8] presented an algorithm running in $O(n \log n / \log \log n)$ time, and later improved this bound to $n2^{O(\sqrt{\log \log n})}$ [9]. 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 running time in a suitable model were given by Buchin and Mulzer [5] and Löffler and Mulzer [14], 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. Radix sort is not only fast in theory but also runs fast in experiments. In contrast, all of the above algorithms for DTs have been only of theoretical interest so far and none of them had been implemented. Many incremental algorithms for DTs like the algorithm by Su and Drysdale [17] use an orthogonal data structure for point location and concepts related to integer sorting to compute these data structures fast. Such algorithms typically run fast in experiments and on random points but have a quadratic worst-case performance.

The goal of our work is to explore ways to obtain a practical algorithm for DTs in the plane that runs in linear time for small integers. Our approach to this is two-fold. First, we implement and evaluate variants of one of the DT algorithms that has the same asymptotic running time as sorting, namely from [5]. Secondly, we consider an algorithm (namely from [1]) and implement and evaluate variants of it, which aim to avoid typical reasons for bad worst-case behavior.

We focus on incremental constructions, more specifically, incremental constructions *con BRIO* [1], where BRIO stands for *biased randomized insertion order*. The points are inserted in random rounds of increasing size and within each round the order of the points can be chosen freely. Amenta, Choi and Rote [1] prove that the expected running time of an incremental construction using such an order is asymptotically bounded by the expected running time of a randomized incremental construction, that is, $O(n \log n)$ if an optimal point location data structure is used[1].

There are various implementations of variants of this algorithm [1, 3, 10, 13, 18]. Most of these variants actually do not use an additional point location data structure and most sort the points of a round along a space-filling curve (SFC). Such variants run in $O(n \log U)$ expected time [4][2], which for small integers is again $O(n \log n)$. In experiments these variants mostly seem to run in linear time, but unfortunately there are point sets for which this bound is tight [4]. Thus if we want an algorithm with a better worst-case performance, we will need to choose a different order.

One weakness of orders based on SFCs seems to be that the construction process does not adapt to the point distribution. In contrast, the CGAL Hilbert curve order [10][3] does. However, this order is likely to introduce some large jumps, that is, 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 the

*Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands.

[1] They actually prove a corresponding result in 3d, but their analysis can be extended to two-dimensional DTs (and other configuration spaces) as shown in [3].

[2] In [4] this bound is formulated in terms of the spread of the point set.

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

This is an extended abstract of a presentation given at EuroCG 2012. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear in a conference with formal proceedings and/or in a journal.

purpose of comparison we also implemented various traditional SFC orders.

The algorithm by Buchin and Mulzer [5] uses an extension of BRIOs that uses nearest neighbor graphs (NNGs) to find a suitable order. While this algorithm does run in linear time in our experiments, the constant factor in the running time is large. The large constant is due to the worst-case optimal NNG construction. Therefore, we implement an additional variant of this algorithm, where we use several steps that are non-optimal but simpler.

2 Algorithms

In this section we discuss several variants of BRIO and BrioDC algorithms that we implemented.

2.1 BRIO

The difference between a regular *random incremental construction* (RIC) algorithm and BRIO, 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 is chosen as an SFC order. By sorting points in this way, the next point to be inserted is likely to be close to the previous point, hopefully resulting in faster point location.

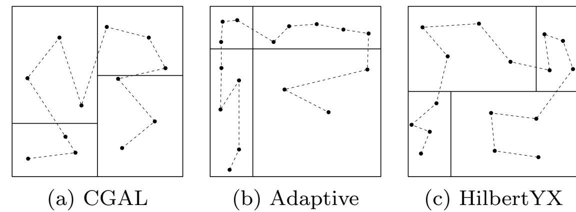
2.1.1 Existing SFCs

There is a plethora of SFCs in the literature and in our experiments, we have used the Peano, Sierpiński and Hilbert curves. In addition, CGAL provides an SFC that is similar to the Hilbert order, but 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 1a. Note that if the two vertical medians differ substantially, the jump from the upper left to upper right quadrant can be large.

2.1.2 New SFCs

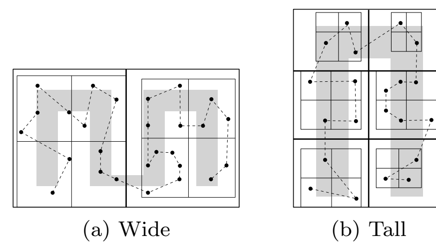
We propose several new SFCs that aim at providing a good mapping between 1-dimensional and d-dimensional space, i.e. no jumps should occur and the total summed distance between consecutive points in the order should be small.

Adaptive Hilbert order. We split the point set by the horizontal and vertical median, as a compromise between the original and CGAL Hilbert orders. This should distribute the points over all quadrants better



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

Figure 1: Variants of the Hilbert SFC.



(a) Wide (b) Tall

Figure 2: Squarified Hilbert SFC.

than the original Hilbert order, and remove the jump that was seen in CGAL Hilbert, see Figure 1b.

Hilbert YX. As a variant on the CGAL Hilbert order, if we first split on the vertical median, and then on the two horizonal medians, no jump occurs when going from one quadrant to another, see Figure 1c.

Squarified Hilbert and Peano orders. If the smallest enclosing bounding box of the point set has large aspect ratio, fitting an SFC 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 SFCs over these point sets. This is achieved by either placing multiple curves next to each other, as in Figure 2a, or using more than four subproblems, as in Figure 2b. The bounding box is recomputed for each subproblem, so the individual subproblems may end up being split in a similar way. Squarifying can also be combined with the adaptive orders.

2.1.3 Implementation

BRIO was implemented in C++ and uses CGAL 3.6.1 to generate the DT after the rounds have been determined. It provides a means to sort a point set according to the SFCs defined in Section 2.1.2. We use CGAL for this part of the experiments to obtain a direct comparison to the CGAL Hilbert order, for which we use the original implementation: `hilbert_sort_2`. The code of CGAL was slightly modified in order to gather metrics.

2.2 BrioDC

BrioDC [5], or BRIO with Dependent Choices, reduces the problem of constructing the DT of a point set P to constructing nearest-neighbor graphs

(NNGs). Let $\mathrm{NNG}_{\leq r}$ be the NNG of the points from the first r rounds. Where BRIO assigns points to a round with equal probabilities, BrioDC makes sure that for every point p that is allocated to round $r > 1$, at least one point q in the connected component of $\mathrm{NNG}_{\leq r}$ is added in an earlier round. We can use the location of q to quickly locate the triangle in which point p should be inserted.

2.2.1 Variations

We use two different algorithms for constructing nearest neighbor graphs.

Linear-time algorithm. In order to find the NNG of a point set, we use a series of intermediate data structures. We start by sorting the point in z-order [15] using radix sort, and from this we generate a *compressed quadtree* using the approach of Chan [7]. The compressed quadtree can be used to find the *well-separated pair decomposition (WSPD)* from which we can compute the NNG using the linear time algorithm of Callahan and Kosaraju [6].

$O(n \log U)$-time algorithm. Although each of the steps in the previous paragraph are done in linear time, the constants in the computation are quite high, so alternatively we have used straightforward implementations that run in near-linear time. The compressed quadtree can be constructed top-down and compressed in $O(n \log U)$ time. For computing the NNG from the WSPD, we use the simpler approach by Har-Peled [12] that runs in $O(n(\log n + \log U))$. Since $\log n$ is bounded by $\log U$, this can be simplified to $O(n \log U)$.

2.2.2 Implementation

We have implemented the BrioDC algorithm in C++. For the base case of the DT and for the incremental insertion of points into the triangulation we use the *Triangle* library[4] [16] version 1.6. In Triangles implementation of an incremental construction of DTs, whenever a point is inserted into the triangulation, Triangle searches for the triangle containing that point. It is possible to supply a guiding triangle, which serves as a start point for the point location query. In each round we use the NNG to supply an appropriate triangle. Other than the modifications to gather the metrics, we have made no changes to the library.

3 Results

We have tested all algorithms on point sets with different distributions; namely uniform (in a square), checkers, bivariate normal with independent coordinates, Kuzmin and line singularity distributions. Checkers is a distribution on a checkers board where the 1/8 of the points are distributed uniformly on white squares and 7/8 of the points are distributed uniformly on black squares. The Kuzmin and line singularity distributions are described by Blelloch et al. [2]. For each distribution and n we run 10 tests on different datasets and report the average of the results. We focus on the running time for the comparison of the algorithms.

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

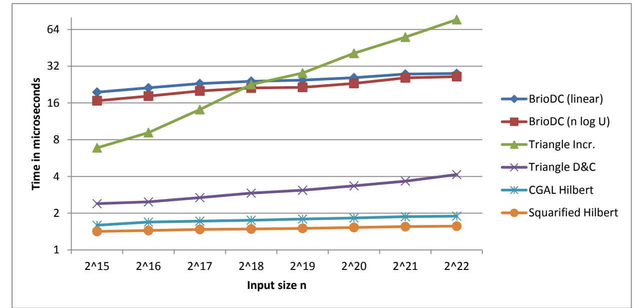


Figure 3: The runtime per point for the different algorithms over all distributions.

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. Only 1 core was used for the experiments.

We first compare the various insertion orders for incremental constructions con BRIO using the running time per point in microseconds for $n = 2^{22}$ over all distributions. The squarified and original Hilbert orders perform best. In general, the squarified orders (Squarified Hilbert 1.57, Squarified Peano 1.58) are slightly slower than the original orders (Hilbert 1.51, Peano 1.57, Sierpiński 1.52). This can be attributed to the higher construction time. Most remaining orders are slower by 25-80% (Adaptive Hilbert 2.01, Adaptive Peano 2.17, HilbertYX 1.95, CGAL Hilbert 1.89, Squarified Adaptive Hilbert 2.71, Squarified Adaptive Peano 2.78), but this drops to 13-25% if we exclude the line distribution (for which all adaptive curves perform poorly).

Next, we compare our algorithms with existing ones. The BRIO orders fall into two categories and the orders in a category have similar running time. Therefore, from the original and squarified orders we only show Squarified Hilbert and from the Adaptive orders we show CGAL Hilbert. Figure 3 shows the running time per point of the algorithms over all distributions on a log-log-scale. Table 1 shows the running times for the different distributions and input size 2^{20}, 2^{21} and 2^{22}. For all algorithms based on con BRIO (including BrioDC) we would expect a constant, or nearly constant running time per point. This seems to be indeed the case, although it increases very slightly with the input size. The running time of the near-linear $O(n \log U)$ implementation of BrioDC is lower than the linear time implementation, but it

Input Size	DCLin	DCLog	TrInc	D&C	CHil	SqHil	Hil
Uniform							
2^{20}	25.56	21.70	35.73	3.06	1.58	1.51	1.46
2^{21}	26.71	24.43	50.46	3.39	1.61	1.53	1.48
2^{22}	28.36	26.01	74.79	3.98	1.64	1.56	1.50
Normal							
2^{20}	25.57	22.96	30.66	3.06	1.60	1.52	1.47
2^{21}	27.33	25.49	41.30	3.34	1.65	1.55	1.49
2^{22}	27.95	27.02	68.21	3.97	1.65	1.55	1.49
Kuzmin							
2^{20}	25.25	23.81	34.27	3.17	1.63	1.55	1.48
2^{21}	29.02	28.09	49.14	3.53	1.67	1.58	1.51
2^{22}	28.75	27.69	70.31	3.98	1.68	1.59	1.51
Checkers							
2^{20}	25.39	22.25	43.65	3.23	1.57	1.50	1.44
2^{21}	27.58	26.45	59.91	3.58	1.61	1.51	1.46
2^{22}	27.07	25.14	74.19	3.94	1.62	1.53	1.47
Line							
2^{20}	26.85	24.86	59.75	4.22	2.76	1.56	1.51
2^{21}	27.13	23.83	76.19	4.49	2.82	1.59	1.53
2^{22}	27.19	25.65	97.93	4.86	2.87	1.60	1.54

Table 1: The runtime per point in microseconds for BrioDC linear (DCLin) and $O(n \log U)$ (DCLog), Triangle incremental (TrInc) and divide-and-conquer (D&C), CGAL Hilbert (CHil), Squarified Hilbert (SqHil) and Hilbert (Hil).

grows 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. Other commonly used $O(n \log n)$-time algorithms come within a factor of 1.5 to 10 of this running time [11]. BrioDC is competitive with these algorithms, and the 5.5-7 factor will decrease further with increasing input sizes. 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 and divide-and-conquer algorithms, and CGAL Hilbert show a large dependency and considerably slow down for the line distribution.

We have also measured the cost of *point location*, *updating the triangulation* and the running time of the *individual parts* of BrioDC. Point location cost is reduced by approximately 3% by using squarified orders compared to the original Hilbert order. The cost for updating the triangulation is similar for all approaches. For BrioDC, about 84% of the running time is spent on computing the NNGs (18% for the quadtree, 29% for the WSPD, and 37% for the NNG from the WSPD).

Summarizing, we have successfully shown the practicality of an $O(n)$-time algorithm for computing the Delaunay triangulation of points with bounded integer coordinates, as well as improved heuristics on non-optimal but faster algorithms. While currently BrioDC is still outperformed by $O(n \log n)$-time algorithms, our implementation is competitive and could be improved upon with a faster NNG algorithm.

Acknowledgements. The authors would like to thank Tal Milea for her valuable contributions.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. 19th Annu. ACM Sym. Comp. Geom. (SoCG)*, pages 211–219. ACM, 2003.

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

[3] K. Buchin. *Organizing Point Sets: Space-Filling Curves, Delaunay Tessellations of Random Point Sets, and Flow Complexes.* PhD thesis, Free University Berlin, 2007.

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

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

[6] 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.

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

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

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

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

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

[12] S. Har-Peled. *Geometric Approximation Algorithms.* Mathematical Surveys and Monographs. AMS, 2011.

[13] Y. Liu and J. Snoeyink. A comparison of five implementations of 3d Delaunay tesselation. In *Comb. and Comp. Geom.*, volume 52 of *MSRI Publications*, pages 439–458. Cambridge University Press, 2005.

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

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

[16] J. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Appl. Comp. Geom. Towards Geometric Engineering*, volume 1148 of *LNCS*, pages 203–222. Springer, 1996.

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

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