Incremental Constructions con BRIO

Nina Amenta [†]
Computer Science Department
University of California at Davis
amenta@cs.ucdavis.edu

Sunghee Choi[‡]
Computer Sciences Department
University of Texas at Austin
sunghee@cs.utexas.edu

Günter Rote
Institut für Informatik
Freie Universität Berlin.
rote@inf.fu-berlin.de

ABSTRACT

Randomized incremental constructions are widely used in computational geometry, but they perform very badly on large data because of their inherently random memory access patterns. We define a *biased randomized insertion order* which removes enough randomness to significantly improve performance, but leaves enough randomness so that the algorithms remain theoretically optimal.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Geometrical problems and computations*

General Terms

Algorithms

Keywords

Randomized incremental construction, Delaunay triangulation, virtual memory.

1. INTRODUCTION

A look at recent textbooks [13, 25] shows that randomized incremental algorithms are a central part of computational geometry. There are many randomized incremental algorithms to build geometric structures; one of particular importance is the randomized incremental construction of the Delaunay triangulation of a set of input points. The algorithm is simple to state: insert the points into the triangulation one by one in random order, updating the triangulation at each insertion. It is also worst-case optimal and (compared to the alternatives) easy to implement robustly. This accounts for its importance in practice; there are several robust and efficient implementations of the randomized incremental Delaunay triangulation construction for $\mathbb{R}^3$, including Clarkson's `hull` [1], the CGAL Delaunay hierarchy function [2], Shewchuk's `pyramid` [3], and in the α-shapes software [4] of Edelsbrunner et al.

Given these excellent programs for three-dimensional Delaunay triangulation, it is natural to want to apply them to the large data sets which arise in applications such as mesh generation or surface reconstruction. By their very essence, randomized incremental algorithms access the geometric data structures randomly, and random access to large data structures works very poorly with modern memory hierarchies. Virtual memory systems cache recently used data in memory, on the assumption of *locality of reference*, that is, that recently used data is likely to be used again soon. Randomized incremental programs violate this assumption, and soon after the data structure exceeds the size of physical memory, thrashing occurs and the program grinds (audibly!) to a halt [9]. This limits the size of the Delaunay triangulations that can be computed in practice.

A simple fix is to insert points in an order which improves the locality of reference, while preserving enough randomness to retain the optimality of the algorithm. In Section 2 we present such a scheme, which we call a *biased randomized insertion order* (BRIO). We prove in Sections 4, 5, 6 that this order is no worse than a completely randomized insertion order in terms of asymptotic complexity: it gives an optimal algorithm for Delaunay triangulation in the worst case, and also under less pessimistic but more realistic assumptions about the output complexity. Our evidence that using a BRIO indeed reduces or eliminates thrashing is experimental. In Section 7 we show the results of using a BRIO with three different three-dimensional Delaunay triangula-

[†]Supported by NSF CAREER award CCR-0093378 and an Alfred P. Sloan Foundation Research Fellowship.
[‡]Supported by grant NSF CCR-0098169

SoCG'03, June 8–10, 2003, San Diego, California, USA.

[1]A program for convex hulls, `http://cm.bell-labs.com/netlib/voronoi/hull.html`
[2]Computational Geometry Algorithms Library `http://www.cgal.org/`
[3]Not yet published
[4]3D alpha shapes software `ftp://ftp.ncsa.uiuc.edu/Visualization/Alpha-shape/`

tion programs: `hull`, which implements the classic textbook randomized incremental construction, the `CGAL` hierarchy program, which uses a different but still optimal algorithm, and `pyramid`, which sacrifices optimality for a small memory footprint. With all of them we can solve much larger problems than were possible with a completely randomized insertion order. Section 9 contains some further discussion of the ideas.

The development of randomized incremental algorithms and their analysis was a major project of computational geometry in the late eighties and early nineties, as described in textbooks [13, 25] and surveys [26, 31]. We touch on a few relevant highlights. A classic paper by Clarkson and Shor [12] showed that the randomized incremental paradigm could be applied to many problems, and gave a general analysis. Mulmuley [22, 23] and Clarkson, Mehlhorn and Seidel [11], among others, extended this theory. Seidel [30], harking back to an an idea in an early paper of Chew [8], popularized a simplifying idea called *backwards analysis* which became the standard tool. Unfortunately we cannot see how to apply backwards analysis when using a BRIO, so we build on results from the earlier work, in particular the bounds on ($\leq k$)-sets from Clarkson and Shor and from Mulmuley.

The traditional approach to thrashing is to develop explicit out-of-core algorithms, usually using divide-and-conquer. Examples in computational geometry include some Delaunay algorithms for 2D [2, 21, 33], which as far as we know are unimplemented. The divide-and-conquer paradigm in general seems to be much less practical than the randomized incremental paradigm in computational geometry. Divide-and-conquer has been useful for providing output-sensitive "theoretical algorithms" (e.g. [7]) but even the relatively simple early divide-and-conquer Delaunay triangulation algorithm of Clarkson [10] seems difficult to implement robustly. An exception is the practical parallel two-dimensional Delaunay triangulation algorithm of Blelloch, Miller, and Talmor [6]. Their approach, however, does not immediately apply to either three dimensions or to out-of-core computation. In this paper, we stick to the randomized incremental paradigm but define an insertion order that, heuristically, helps the memory hierarchy work effectively.

2. INSERTION ORDER

We define a *biased randomized insertion order* for a set P of n points. The points are inserted in *rounds*, from round 0 through round $\lg n$. For simplicity, we assume that n is a power of 2.

To allocate points to rounds, we choose each point independently with probability 1/2 to be inserted in the final round. We choose each of the remaining points independently with probability 1/2 to be inserted in the next-to-last round, and so on. When we get to round zero, we choose any remaining points with probability one. Thus the probability that a point is chosen in round $i > 0$ is $2^{i-1}/n$, and the remaining probability $1/n$ goes to the event that the point is chosen in round zero. As far as our proof of asymptotic optimality is concerned, the points can be inserted in an arbitrary order within each round.

This restricted requirement on the randomness allows us to bias the insertion order to favor locality. The approach we take here is to organize the points into *blocks* which respect locality within three-dimensional space; in our experiments we used either an oct-tree or a *kd*-tree (e.g., [13]) to divide space into blocks. Within each round, we group the points by block, and we order the blocks themselves within a round to favor locality in by taking them in depth-first order. Within each block we order the points randomly.

The intuition is that in the early rounds the insertions tend to be sprinkled nearly randomly across all the data, producing a nicely balanced data structure, while in the later rounds they are grouped by blocks, accessing local regions of the data structure mostly independently.

Notice that our ordering, which improves locality in $\mathbb{R}^3$, only indirectly attacks the question of locality in the layout of the data structure in virtual memory, which is the essential problem. The hope is that inserting points with locality in $\mathbb{R}^3$ encourages locality in virtual memory. However, this depends on the storage management scheme used by the specific Delaunay program, and may be hard to predict. We return to this issue when we discuss the experiments.

3. ANALYSIS SETUP

We analyze the use of a BRIO in the context of the incremental construction of a three-dimensional Delaunay triangulation. First, we review the analysis of the usual randomized incremental algorithm for three-dimensional Delaunay triangulation ([12, 11] or see [13, 25]). The running time can be divided into two parts, the time required to find where each new point should be inserted into the Delaunay triangulation (*point location time*) and the time required to delete old tetrahedra and create new tetrahedra so as to actually perform the insertion (*update time*). Point location can be done in various ways; the theoretically optimal methods have been shown to be $\Theta(X(n))$, where $X(n)$ is a quantity known as the *total conflict size*.

Total update time is $\Theta(C(n))$, where $C(n)$ the total number of tetrahedra which appear over the course of the construction.

3.1 The "realistic" case.

In the worst case, the size of a Delaunay triangulation of n points in $\mathbb{R}^3$ is $O(n^2)$, and it turns out that this is also the bound on the total conflict size and hence the running time. But in practice the size of the Delaunay triangulation is often $O(n)$.

We make a slightly stronger assumption: in the "realistic" case the expected size of the Delaunay triangulation of a random sample of r of the points is $O(r)$. That is, we define a "realistic" instance to be a point set P for which the Delaunay triangulation of a random subset of r points has expected size $O(r)$, for every r. Experiments with data from various sources [9] give evidence that many instances are in fact "realistic".

In the "realistic" case there is a more realistic bound of $O(n \lg n)$ on the total conflict size and the running time for the standard randomized incremental construction. We show below that the algorithm remains optimal using a block randomized insertion order in the worst case, and also in this "realistic" case.

More generally, we can consider a probability distribution of "practical" instances P, and the $O(r)$ size requirement should hold for a random (uniform) r-subset of a random problem instance (according to the given distribution). Our results about expected running time are then average-case results (in addition to the expectation with respect to the random choices of the algorithm).

3.2 The general framework.

We will use some terminology of Mulmuley. The four vertices of a tetrahedron are known as its *triggers*, and the other points of P contained in its circumsphere are called its *stoppers*. Every choice of four points of P as triggers determines a possible tetrahedron, but in a particular run of the randomized incremental construction not every possible tetrahedron *appears* as part of one of the intermediate triangulations, or in the final triangulation. A tetrahedron appears in some Delaunay triangulation if and only if all of its triggers are selected for insertion before any of its stoppers. The probability that a tetrahedron appears during the construction thus depends in part on its number s of stoppers; if $s = 0$, for instance, the tetrahedron belongs to the final Delaunay triangulation and the probability that it appears is one.

The structure of the analysis follows the scheme in the early papers [12, 23, 24]. Let p_s be the probability that a tetrahedron with s stoppers appears in some triangulation of the construction, and let k_s be the number of tetrahedra with s stoppers for point set P. Then we can write the expected total number of tetrahedra that appear as

$$E[C(n)] = \sum_{s=0}^{n} k_s p_s$$

The total conflict size is the sum, over all tetrahedra τ which ever appear in the construction, of the number of stoppers of τ. Thus, the expected total conflict size is

$$E[X(n)] = \sum_{s=0}^{n} k_s p_s s.$$

Now all we need are upper bounds on k_s and p_s.

4. ONE TETRAHEDRON

Let us consider a tetrahedron τ with s stoppers. If s is zero, τ has to appear in the Delaunay triangulation. If s is non-zero *and* τ appears in some intermediate triangulation, inevitably in some later insertion one of τ's stoppers will be chosen and τ will be *popped* (it will no longer be part of the triangulation). In other words, the probability that τ appears is the same as the probability that τ is eventually popped.

We bound the probability p_s that τ appears by considering each round i of the BRIO, building on the following.

Observation 1. *The probability that a point $x \in P$ is chosen in round i is $2^{i-1}/n$ for $i \geq 1$ and $1/n$ for $i = 0$. Each point is assigned to a round independently.*

Lemma 2. *A tetrahedron τ with s stoppers appears with probability $p_s = O(1/s^4)$.*

Proof. If τ is popped in round i, it must be case that all triggers of τ were chosen in or before round i and that the first stopper is chosen in round i. Since these events are independent, the probability that τ appears is bounded as follows.

$$p_s \leq \sum_{i=0}^{\lg n} P[\text{ first stopper in round } i\text{ }] \times P[\text{ all triggers in round } i \text{ or earlier }] \quad (1)$$

Let a_i be the probability that a particular point has been chosen at the beginning of round i, so that

$$a_0 = 0,\ a_1 = 1/n,\ a_2 = 2/n,\ a_3 = 4/n,\ \ldots,\ a_{\lg n+1} = 1,$$

and in general $a_i = 2^{i-1}/n$. Then we have

$$a_{i+1} = 2a_i \quad (2)$$

for $i > 0$. The probability that all of the triggers are chosen in or before round i is a_{i+1}^4 since the events are independent.

The probability that the first stopper is selected in round i is denoted by $G_i = (1 - a_i)^s - (1 - a_{i+1})^s$. By (1), the probability that τ appears is

$$p_s \leq \sum_{i=0}^{\lg n} G_i \cdot a_{i+1}^4.$$

We split off the first term, and for the remaining terms we use (2).

$$p_s \leq 1/n^4 + \sum_{i=1}^{\lg n} G_i \cdot 16 a_i^4$$

To bound the latter sum, note that $G_i \cdot a_i^4$ can be interpreted as the probability of another event: the first stopper is chosen in round i, and at the beginning of the i-th round, all of the triggers have been chosen. Thus $\sum_{i \geq 1} G_i \cdot a_i^4$ is the probability that the round in which all of the triggers are chosen is smaller than the first round where any stopper is chosen. This is the probability that among $s + 4$ independent identically distributed random variables (namely the rounds of the 4 triggers and the s stoppers), the first 4 are (strictly) smaller than the others. By symmetry, we get

$$\sum_i G_i \cdot a_i^4 \leq \frac{1}{\binom{s+4}{4}}.$$

(This is the same argument that applies to the "usual" (fully) randomized insertion order.) This gives us

$$p_s \leq 1/n^4 + \frac{16}{\binom{s+4}{4}} = O(1/s^4). \qquad \square$$

5. COUNTING TETRAHEDRA

Now we need to bound k_s, the number of tetrahedra with s stoppers. Let K_s be the number of tetrahedra with *at most* s stoppers. Clarkson and Shor [12, Section 3] gave an upper bound on ($\leq k$)-sets that implies that that K_s is at most $O(n^2 s^2)$ in the worst case, and $O(ns^3)$ in the "realistic" case. The bound was proved in a different way by Mulmuley [24]. In this section we give proofs along the same lines as Mulmuley's but with simpler arithmetic; this is possible because we deal only with the special cases of K_s that we need.

Consider the following thought experiment. From the set P of n points, we select each point with probability $1/s$ to form a random sample R. Let $r = |R|$ be the random variable for the size of R. Let T_R be the random variable for the number of tetrahedra in the Delaunay triangulation of R.

Lemma 3 ([12, Theorem 3.1]). *For every point set P, we have*

$$K_s = O(s^4 E[T_R]).$$

As a consequence of this lemma, we get the following bounds on K_s.

In the "realistic" case, we assume that $E[T_R] = O(r)$ so that by the linearity of expectation $E[T_R] = O(E[r]) = O(n/s)$, and $E[K_s] = O(ns^3)$.

In the quadratic worst case, we have $T_R = O(|R|^2)$, and hence $E[T_R] = E[O(r^2)] = O(E[r^2])$.

$$E[r^2] = \mathrm{Var}[r] + E^2[r] = n\left(\frac{1}{s}\right)\left(1-\frac{1}{s}\right) + (n/s)^2 = O((n/s)^2)$$

So, $K_s \leq O(s^4(n/s)^2) = O(n^2s^2)$.

For completeness, we indicate the easy proof of the lemma. We assume without loss of generality that $s \geq 2$. Let $\tilde{p}_j$ denote the probability that a tetrahedron with j stoppers appears in the Delaunay triangulation of R. For $j \leq s$,

$$\tilde{p}_j = \left(\frac{1}{s}\right)^4\left(1-\frac{1}{s}\right)^j \geq \left(\frac{1}{s}\right)^4\left(1-\frac{1}{s}\right)^s = \Theta\left(\frac{1}{s^4}\right)$$

and therefore $\tilde{p}_j = \Theta(1/s^4)$. We can express $E[T_R]$ in another way:

$$E[T_R] = \sum_{j=0}^{n} \tilde{p}_j k_j \geq \Theta\left(\frac{1}{s^4}\right)\sum_{j=0}^{s} k_j = \Theta\left(\frac{1}{s^4}\right)K_s$$

From this, the lemma follows. □

6. RUNNING TIME

THEOREM 4. *With incremental construction using a BRIO, the expected total number of tetrahedra that are created during the construction of the Delaunay triangulation of n points in three dimensions is $O(n^2)$ in the worst case and $O(n)$ in the realistic case. The expected total conflict size (and hence the expected running time) is $O(n^2)$ in the worst case and $O(n \log n)$ in the realistic case.*

Proof. Recall that our expression for the expected total number of tetrahedra is

$$E[C(n)] = \sum_{s=0}^{n} k_s p_s$$

and that $p_s = O(1/s^4)$, and note that $p_0 = 1$. We choose a constant c such that $p_s \leq c/s^4$, for $s \geq 1$. So

$$\begin{aligned} E[C(n)] &\leq K_0 + \sum_{s=1}^{n}(K_s - K_{s-1})\,\frac{c}{s^4} \\ &= (1-c)K_0 + c\cdot\sum_{s=1}^{n-1} K_s\left(\frac{1}{s^4} - \frac{1}{(s+1)^4}\right) + \frac{c\,K_n}{n^4} \\ &= O(K_0) + \sum_{s=0}^{n-1} O\left(\frac{K_s}{s^5}\right) + O\left(\frac{K_n}{n^4}\right) \end{aligned}$$

In the "realistic" case, $K_s = O(ns^3)$ so $E[C(n)] = O(n)$. In the worst case $K_s = O(n^2s^2)$ and we find that $E[C(n)] = O(n^2)$.

We can use a similar argument to bound the total conflict size and hence the point location time.

$$E[X(n)] = \sum_{s=1}^{n} k_s \frac{c}{s^4}\cdot s = O(K_0) + \sum_{s=1}^{n-1} O\left(\frac{K_s}{s^4}\right) + O\left(\frac{K_n}{n^3}\right)$$

This gives $E[X(n)] = O(n \log n)$ in the "realistic" case and $E[X(n)] = O(n^2)$ in the worst case. □

7. EXPERIMENTS

We have tried using biased randomized insertion orders with three different 3D Delaunay triangulation codes, Clarkson's `hull`, the `CGAL` Delaunay hierarchy function and Shewchuk's `pyramid`. These programs differ in their point location schemes and memory management strategies. Nonetheless, with all the programs, using the BRIO produced a near-linear performance profile, allowing us to handle much larger inputs than we could with a completely randomized insertion order.

While we of course had hoped that the effect of increased locality in $\mathbb{R}^3$ on the performance would be beneficial, it is not easy to predict. A fundamental problem with trying to optimize memory access by increasing the locality of insertions in $\mathbb{R}^3$ is that a point might well share Delaunay edges with other points that are quite far away. In particular, for sets of input points which lie on or close to surfaces (important to applications such as surface reconstruction and mesh generation) every point usually has at least one edge to some vertex on a different "opposite" part of the surface. Our experiments use this kind of input data.

Moreover, since the Delaunay triangulation is represented by a pointer structure, there is no guarantee that adjacent tetrahedra in the triangulation are stored together in virtual memory; this is implementation dependent. All three of the programs do their own memory management for the tetrahedra, to avoid making too many calls to the memory allocation procedure of the operating system. Records for tetrahedra are stored in a list in virtual memory. In the basic randomized incremental construction implemented by `hull`, tetrahedra are never deleted, but in `pyramid` and `CGAL` they are. Records are freed as tetrahedra are destroyed and reused as new tetrahedra are created, so that a tetrahedron might end up being stored with others created much earlier, further reducing locality.[5]

In all of our experiments we used a large (4 GB) virtual memory, which on our systems required some reconfiguration. This is important for duplicating our results; the programs fail if they run out of virtual memory. The idea of using a BRIO is to avoid having to explicitly manipulate disk access and letting the (hopefully very efficient) virtual memory system do the work.

7.1 Data sets

We used two data sets in the experiments. Both of them are on or near a 2D surface in $\mathbb{R}^3$, since that case is important in practice and the Delaunay triangulation is non-local, as described above.

To make the B1 data set (525,296 points), we extracted a molecular electron density iso-surface using marching cubes and then made it larger by applying one level of butterfly subdivision. This gives a nicely distributed point set which lies on a smooth surface.

The happy buddha data set (2,643,633 points) is taken from the Stanford 3D scanning repository [6]. We use the raw scanner data as an example of typical input to a surface reconstruction computation. The data is noisy and unevenly

[5]We thank Jonathan Shewchuk for pointing out this issue.

[6]Stanford 3D scanning repository, `http://www-graphics.stanford.edu/data/3Dscanrep`

distributed near the surface of the object.

We concentrated in these experiments with using different insertion orders and different programs rather than many data sets; it would be interesting to try similar experiments with other data, especially data from different kinds of distributions.

7.2 Randomized incremental construction

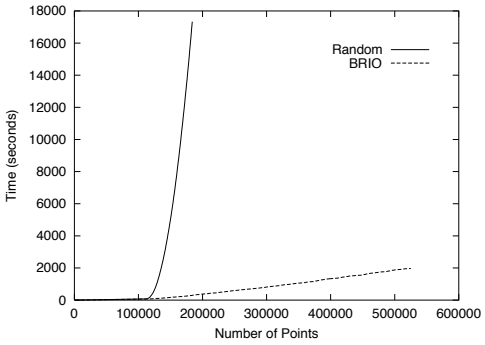

Figure 1: The running time of `hull` on B1 data using a completely randomized insertion order and a BRIO (512 MB RAM).

To test exactly the situation considered in our analysis, we used a BRIO with the standard randomized incremental construction as implemented in `hull`. This program uses a theoretically optimal point location data structure, the history DAG, so that the expected point location time is proportional to the total conflict size. Also, since no tetrahedra are ever deleted, the layout of tetrahedra in virtual memory should correspond well to the order in which they are created. The history DAG takes a lot of memory, however, so `hull` thrashes relatively early.

We ran `hull` on a Linux machine with an Intel Pentium III (864 MHz) and 512 MB RAM, using the smaller B1 data.

In Figure 1, the running time for `hull` using the completely random insertion order and the BRIO are shown; the random insertion order led to thrashing before the triangulation could be completed. Though the slope for the biased randomized insertion order becomes steeper around 120K points—just where there was a serious thrashing for random order—the BRIO maintains a roughly constant slope and shows a near-linear running time.

7.3 CGAL hierarchy

The Delaunay hierarchy function in the `CGAL` library also implements an optimal algorithm, due to Devillers [14], based on a data structure for point location which requires much less memory than the history DAG. Deviller's analysis depends on the randomized insertion order to bound the expected point location time, but does not explicitly relate the running time to the total conflict size. We hope to show that this data structure is optimal when using a BRIO in the journal version of the paper.

To exaggerate the memory behavior of the program, we ran it on a very small PC: a Sun UltraSPARC 360 MHz CPU with just 128M of physical memory. With so little memory, the `CGAL` Delaunay hierarchy program begins to thrash at about 250K points when using a completely randomized insertion order. Using a BRIO (Figure 2) we compute the Delaunay triangulation of the happy buddha data set containing more than 2.5 million points.

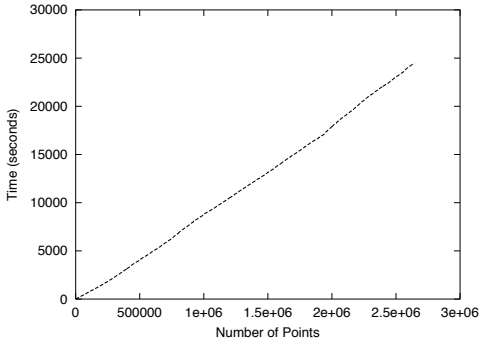

Figure 2: Running time of the `CGAL` Delaunay hierarchy on the happy buddha scan data using a BRIO (128 MB RAM). Using a randomized insertion order with this small memory, the program would thrash at about 250K points.

7.4 Pyramid

Shewchuk's `pyramid` is designed to be very memory efficient. It uses a theoretically non-optimal point location scheme but needs no additional storage beyond the Delaunay triangulation itself. As described above, the deletion of tetrahedra and the re-use of their virtual memory complicates the layout in virtual memory. Our analysis shows that when using a BRIO the total number of tetrahedra created is asymptotically optimal, but other than that the relationship to the theory is tenuous in the case of this program. The fact, however, that we can compute huge 3D Delaunay triangulations, very quickly, using very little physical memory, is exciting.

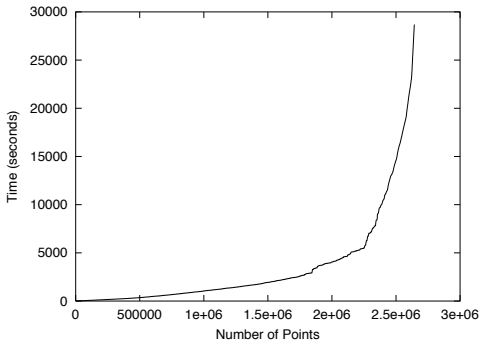

Figure 3: The running time of `pyramid` on the happy buddha data using BRIO and `pyramid`'s original point location scheme (128 MB RAM).

We again use the small PC and the happy buddha data. We were able to complete the Delaunay triangulation of the happy buddha using the BRIO, which was not possible with the completely randomized insertion order. But we found that as the size of the data structure grew, `pyramid`'s point location strategy slowed down significantly (Figure 3). We had never seen this effect before because we had never been able to compute such a large triangulation.

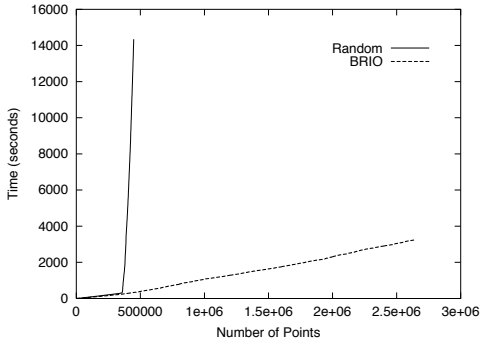

Figure 4: The running time of `pyramid` with the simplified point location strategy, on the happy buddha data set using a BRIO and a completely randomzied insertion order for comparison (128 MB RAM).

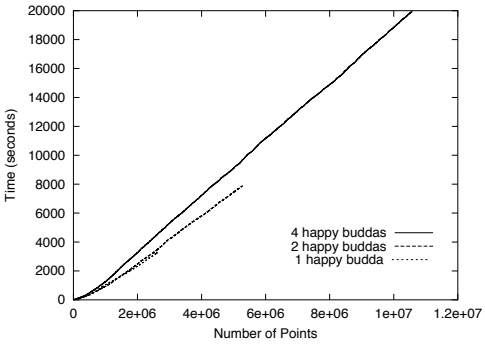

Figure 5: The running time of `pyramid` on 1,2 and 4 happy buddha data sets (128 MB RAM).

The point location strategy used in `pyramid` is known as jump-and-walk. At the insertion of the ith point p, it selects $O(i^{1/4})$ already-inserted points at random, and finds the closest of these to p. It then selects either this point, *or the last point inserted*, whichever is closer to p, and begins "walking" in the Delaunay triangulation on a straight line from there to find the place at which to insert p. With the BRIO, the last point inserted was almost always the closest to p, and the expensive search for a closer point was generally wasted.

Eliminating the $O(i^{1/4})$ search and always starting from the last point inserted gave us an essentially linear running time (Figure 4). To see how far we could push the results, we duplicated and translated the buddha data, making inputs that were the union of two and of four buddhas. We found that we could complete the Delaunay triangulation of four buddhas, over 10 million points (Figure 5). This represents an increase by a factor of 20 in the size of the Delaunay triangulations computable on this machine with this program.

Using this small machine is useful for studying memory performance, but it makes Delaunay triangulation look slower than it really is. On the machine we used for the experiments with `hull`, we can compute the Delaunay triangulation of 10 million points in about half an hour (Figure 6).

Of course, with this point location strategy there are no theoretical bounds on the expected running time besides

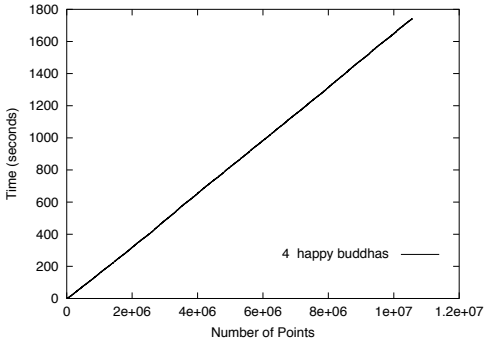

Figure 6: The running time of `pyramid` on a more typical workstation (864 MHz, 512 MB RAM). We computed the Delaunay triangulation of 10 million points (four translates of the happy buddha scan data) in about half an hour, using a BRIO with the simplified point location scheme.

the pessimistic worst case of $O(i)$, for a total running time of $O(n^2)$. However, for reasonable point sets, and for good insertion orders within each round, it seems reasonable to expect a good performance. For uniformly distributed point sets, a simple algorithm with linear expected running time has been proposed by Dyer [17]. It is conceivable that our approach might lead to an alternate algorithm for uniformly distributed point sets which degrades gracefully as the uniform distribution assumption is violated.

7.5 Pure locality

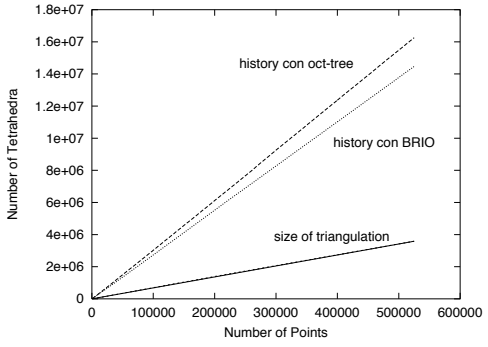

Figure 7: Total number of tetrahedra in the history DAG, for `hull` using the B1 data set. The bottom curve is the number of tetrahedra in the actual Delaunay triangulation.

The locality of the BRIO certainly seems to improve the memory performance. To see if the randomness in the BRIO provides any practical benefit, we compared the three programs using a BRIO and an order which visits each oct-tree cell in turn and inserts all of its points in random order. We found that both orders worked well, but which is better varied.

Figures 7, 8 show an example using `hull`. The BRIO creates a slightly smaller history DAG. Before physical memory is exceeded, the BRIO does better, but once it begins paging the better locality of the oct-tree order dominates.

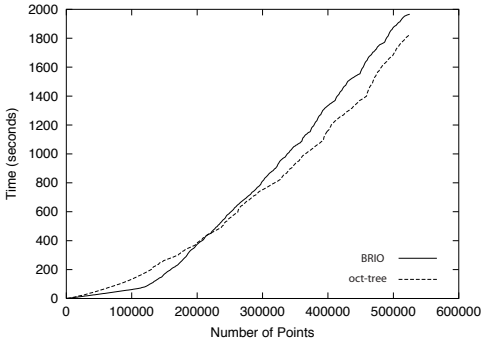

Figure 8: The oct-tree insertion order with `hull` runs slightly faster on the B1 data, although the BRIO creates fewer total tetrahedra (512 MB RAM).

Running the larger data set with `CGAL`, we found that the BRIO gave a slightly better running time (Figure 9); here again possibly the point location data structure is better with the BRIO.

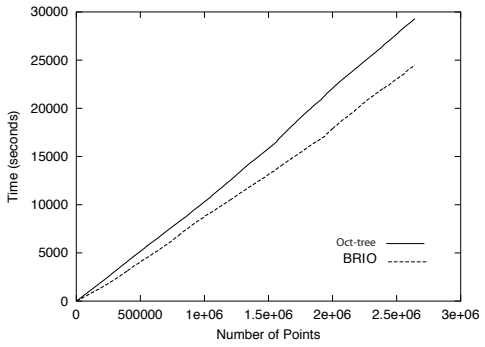

Figure 9: Using a BRIO gives a slightly better running time for the happy budda data using `CGAL` (128 MB RAM).

Using pyramid, on the other hand, we found that the purely local order was better(Figure 10).

It is hard to draw conclusions from these examples, other than to note that factors like the specific memory hierarchy, layout in virtual memory, number of tetrahedra created and destroyed, balance in the point location data structure, etc., interact in complicated ways.

8. COMPUTING A BRIO

Using a BRIO to improve the performance of Delaunay triangulation programs would not be sensible if computing the BRIO itself was time consuming compared with the time required to shuffle points for the randomized incremental construction. Fortunately, BRIOs can be computed very efficiently. [7]

We tried both *kd*-trees and oct-trees for decomposing the input point sets into blocks for creating BRIOs. The *kd*-tree construction is appealing since it has logarithmic depth,

[7]We thank Yong Kil (UC Davis) for his contribution to this section, including his implementation of the oct-tree BRIO computation and his ideas for optimizing it.

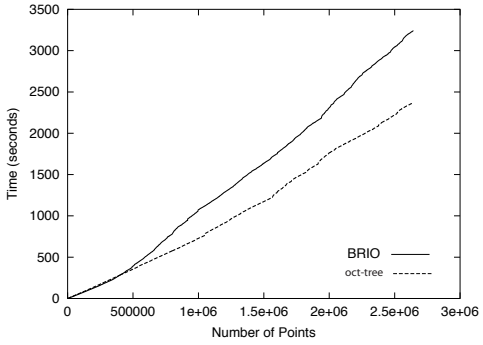

Figure 10: The purely local oct-tree order runs faster than the BRIO for the happy budda data with `pyramid`, and the simplified point location scheme (128 MB RAM).

and we can guarantee that every block has roughly the same number of points. However, neither of these properties are actually required, and constructing a *kd*-tree requires sorting while constructing an oct-tree does not. Interestingly, the standard UNIX sort we used thrashed on our smallest (128 MB) machine. Since the performance of the two kinds of BRIOs were essentially the same, we prefer the oct-tree.

For all the BRIOs, we used an upper limit of 2000 points per block. Our experience has been that block size should be a few thousand, but the exact number does not seem to make much difference.

After computing the oct-tree, the points are randomly shuffled within each block. In round i, we visit each block in turn. Within each block, we visit each uninserted point and insert it with the appropriate probability for the round. The overhead for visiting each point in each round is small compared to the time required to compute the oct-tree.

The entire computation is extremely fast. We can compute the BRIO on 10 million points in about 2 minutes, including the computation of the oct-tree (on a 1.7 GHz Pentium IV with 512 MB RAM).

9. DISCUSSION

There are a number of questions raised by this research. It would be interesting to find a point location strategy which is theoretically optimal but constant time in practice given some BRIO, or even constant time in theory given some (easily computable) BRIO.

The biasing scheme outlined in this paper is simply the first one that we thought of. There are many others possible ways of defining blocks, or a BRIO might be based on some fixed ordering of the points that respects locality, such as a space-filling curve [3, 5, 27, 28]. Schemes that take into account the layout of the data structure in virtual memory instead of in three-dimensional space, and adaptive schemes which use the results of early insertions to order later insertions might be particularly effective or theoretically interesting.

Although we give the proofs in terms of the Delaunay triangulation construction, the analysis applies to other similar randomized incremental constructions, in particular the optimal construction of the trapezoidation of a set of non-intersecting segments in the plane [12, 22, 23], and the sim-

ilar construction for intersecting segments. A drawback of the analysis in this paper, as opposed to backwards analysis, is that it depends on every object having the same number of triggers. For the trapezoidation algorithm, this forces us to again follow the older analysis [22] and count the number of *attachments* (the vertical faces of trapezoids) rather than the trapezoids themselves. The analysis in this paper must be modified to account for the fact that objects have different numbers of triggers. As long as the number of triggers is bounded by some constant, we can simply analyze the objects separately for each possible number of triggers.

We believe that this analysis can also be applied to the related randomized incremental algorithms which use *tracing*, such as Seidel's practical $O(n \lg^* n)$ algorithm for trapezoidation of a simple polygon [29]. BRIOs might also work with the LP-type (also known as GLP, "generalized linear programs") randomized incremental algorithms, which optimize an objective function over a set of input regions. This might not be very interesting, however, since LP-type algorithms do not build large data structures.

On the other hand, the performance of LP-type algorithms can be enhanced in other ways by heuristic insertion orders [32]. Similarly Barber's `qhull` program for arbitrary-dimensional convex hull uses a heuristic insertion order designed to insert points on the convex hull early [4]. Particular biased randomized insertion orders, or some other partially-random scheme, might allow these heuristics to be applied while still maintaining optimality.

Devillers and Guigue [15] considered a different kind of partially randomized insertion order, for handling constructions for which the data is provided sequentially rather than all at once. Arriving data can be stored and reshuffled (randomly) in a buffer of limited size before it has to be inserted into the data structure. They showed that the expected running time degrades as a smaller shuffling buffer is used and the randomness is more limited. For an analysis similar to ours to work, it seems important to have at least a random sample of all of the points available at the beginning of the construction. The following example illustrates this point.

Consider a set P of points in $\mathbb{R}^3$ distributed uniformly at random on two squares, R_1 and R_2, of equal size, lying on two parallel planes opposite to each other. Golin and Na [19, 20] showed that not only the whole set, but also any large enough random subset has a linear-size Delaunay triangulation, so this is an example of the "realistic" case. In the completed Delaunay triangulation each point belongs to a constant number of edges, on average. Some edges connect it to its neighbors on the same square and some connect it to "opposite" points on the other. Now assume that the points are provided sequentially, with all the points on R_1 given first, and then the points on R_2, in order of distance from the left edge. Each new point on R_2 forms tetrahedra with its "opposite" points, and also with points on the right side of R_1 whose "opposite" points on R_2 have not yet been added; as soon as any points are added on R_2, there are edges going from R_2 to all the points on R_1. The points on the right side of R_1 are always connected to the right-most points on R_2. Since these right-most points are constantly being replaced, the total number of tetrahedra created ends up being $\Theta(n^{3/2})$.

10. REFERENCES

[1] Agarwal, P. K., de Berg, M., Matoušek, J., and Schwarzkopf, O. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM J. Comput.* **27** (1998), 654–667.

[2] Arge, L. External memory data structures. In *Handbook of Massive Data Sets*, J. Abello, P. M. Pardalos, and M. G. C. Resende, Eds., vol. 4 of *Massive Computing*. Kluwer Academic Publishers, 2002, ch. 9, pp. 313–357.

[3] Asano, T., Ranjan, D., Roos, T., Welzl, E., and Widmayer, P. Space-filling curves and their use in the design of geometric data structures. *Theoret. Comput. Sci.* **181**, 1 (July 1997), 3–15.

[4] Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.* **22**, 4 (Dec. 1996), 469–483.

[5] Bartholdi, III, J. J., and Platzman, L. K. A fast heuristic based on spacefilling curves for minimum-weight matching in the plane. *Inform. Process. Lett.* **17** (1983), 177–180.

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

[7] Chan, T. M., Snoeyink, J., and Yap, C. K. Primal dividing and dual pruning: Output-sensitive construction of 4-d polytopes and 3-d Voronoi diagrams. *Discrete Comput. Geom.* **18** (1997), 433–454.

[8] Chew, L. P. Building Voronoi diagrams for convex polygons in linear expected time. Technical Report PCS-TR90-147, Dept. Math. Comput. Sci., Dartmouth College, Hanover, NH, 1986.

[9] Choi, S., and Amenta, N. Delaunay triangulation programs on surface data. In *Proc. 13th Ann. ACM-SIAM Symposium on Discrete Algorithms, January 6–8, 2002, San Francisco* (2002), ACM/SIAM, pp. 135–136.

[10] Clarkson, K. L. New applications of random sampling in computational geometry. *Discrete Comput. Geom.* **2** (1987), 195–222.

[11] Clarkson, K. L., Mehlhorn, K., and Seidel, R. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.* **3**, 4 (1993), 185–212.

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

[13] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[14] Devillers, O. The Delaunay hierarchy. *Int. Journ. Found. of Comp. Sci.*, **13** (2002), 163–180.

[15] Devillers, O., and Guigue, P. The shuffling buffer. *Internat. J. Comput. Geom. Appl.* **11** (2001), 555–572.

[16] Dwyer, R. A. *Average-case analysis of algorithms for convex hulls and Voronoi diagrams*. Ph.D. thesis, Comput. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, PA, 1988. Report CMU-CS-88-132.

[17] Dwyer, R. A. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete Comput. Geom.* **6** (1991), 343–367.

[18] Golin, M. J., and Na, H.-S. On the average complexity of 3d-Voronoi diagrams of random points on convex polytopes. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.* (2000), pp. 127–135.

[19] Golin, M. J., and Na, H.-S. On the average complexity of 3d-Voronoi diagrams of random points on convex polytopes. Tech. Rep. Tech. report HKUST-TCSC-2001-08, Hong Kong University of Science and Technology, 2001. `http://www.cs.ust.hk/tcsc/RR/2001-08.ps.gz`.

[20] Golin, M. J., and Na, H.-S. On the proofs of two lemmas describing the intersections of spheres with the boundary of a convex polytope. Tech. Rep. Tech. report HKUST-TCSC-2001-09, Hong Kong University of Science and Technology, 2001. `http://www.cs.ust.hk/tcsc/RR/2001-09.ps.gz`.

[21] Goodrich, M. T., Tsay, J.-J., Vengroff, D. E., and Vitter J. S. External-Memory Computational Geometry, In *Proc. 34th Foundations of Computer Science (FOCS)* (1993), pp. 714–723.

[22] Mulmuley, K. A fast planar partition algorithm, I. *J. Symbolic Comput.* **10**, 3-4 (1990), 253–280.

[23] Mulmuley, K. A fast planar partition algorithm, II. *J. Assoc. Comput. Mach.* **38** (1991), 74–103.

[24] Mulmuley, K. On levels in arrangements and Voronoi diagrams. *Discrete Comput. Geom.* **6** (1991), 307–338.

[25] Mulmuley, K. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[26] Mulmuley, K., and Schwarzkopf, O. Randomized algorithms. In *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, Eds. CRC Press LLC, Boca Raton, FL, 1997, ch. 34, pp. 633–652.

[27] Platzman, L. K., and Bartholdi III, J. J. Spacefilling curves and the planar travelling salesman problem. *J. Assoc. Comput. Mach.* **39** (1989), 719–737.

[28] Sagan, H. *Space-filling curves*. Springer-Verlag, New York, 1994.

[29] Seidel, R. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.* **1**, 1 (1991), 51–64.

[30] Seidel, R. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.* **6** (1991), 423–434.

[31] Seidel, R. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*, J. Pach, Ed., vol. 10 of *Algorithms and Combinatorics*. Springer-Verlag, 1993, pp. 37–68.

[32] Welzl, E. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, H. Maurer, Ed., vol. 555 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1991, pp. 359–370.

[33] Zhu, B. Further computational geometry in secondary memory, In *5th Annual International Symposium on Algorithms and Computation (ISAAC)* (1994), pp. 514–522.