The Shuffling Buffer

Olivier Devillers[1] Philippe Guigue[1]

1 Introduction

The classical approach of computational geometry is the search for algorithms having the best possible worst case complexity, unfortunately, these algorithms are often fairly complex and difficult to implement. An attractive alternative is to use simpler randomized algorithms, whose complexities are not worst case optimal, but optimal only when averaging over all the possible executions induced from some random choices done by the algorithm.

From a given input, a classical algorithm has a single deterministic way to go to the solution while a randomized algorithm chooses at random between different ways going from the input to its solution.

Several methods exist to introduce randomness in an algorithm [12, 10, 3, 1]. One possibility is to use an incremental algorithms whose complexity depends on the order in which input is presented to them. Then, a randomized analysis assumes that all the $n!$ possible orders to introduce the n data are equally likely. Previous application of a random permutation then replaces (typically *bad*) worst-case times by (typically *good*) expected time behaviour. However, the advantage of this approach - simple algorithms, improved by means of coin flips - has to be payed for because now, the whole input must be known in advance, contradicting the *on line* incremental paradigm.

Thus, we have to face the alternative: wait for all data and then start the incremental construction with a random order, or start the algorithm on-line and hope that the data are in a sufficiently random order. The *Shuffling Buffer* provides a straight-forward way to trade-off between both extremes: partition the input stream into blocks of size k; randomly permute the elements *within* each block but don't change the order of the blocks themselves. This technique permits to introduce enough randomness to guarantee some improvement on the worst case complexity without modifying the underlying algorithm.

This paper illustrates the shuffling buffer method for sorting, the Delaunay triangulation and trapezoidal map. However, we would like to point out that this kind of randomized technique is suitable in many other domains.

Overview We start with presenting a simple sorting algorithm (Section 2) as a typical example of a *randomizable* incremental and order dependant algorithm. The *shuffling buffer model* is described together with its analysis.

Following this, we turn our attention to the problem of constructing the Delaunay triangulation of a set of points in the plane (Section 3) or determining all intersection pairs among a set of line segments in the plane (Section 4).

The guarantee of an improvement of a $\frac{\log k}{k}$ factor with respect to their worst case complexity is proved for both algorithms upon use of a buffer of size k.

Section 3.4 then provides experimental results for two and three dimensional Delaunay triangulation on a practical application. In this application, the order is given by the acquisition process. Changing it with the shuffling buffer technique provides better results.

[1]http://www-sop.inria.fr/prisme/

2 A simple example: sorting

For the particular problem of sorting, there exists deterministic (balanced binary tree) as well as randomized (skip-list) solutions which do not need a random order.

The presented scheme is a sorting algorithm with total sorting time $\Theta(n^2)$ in the worst case on a n numbers input. In spite of this bad running time in the worst case, this algorithm is often a good choice in practice because of its notable efficiency on average: the expected running time is $\Theta(n \log n)$.

2.1 Description of the algorithm

The presented sorting algorithm is very simple and consists of inserting each number of a set of n numbers one by one in an usual binary search tree without balancing scheme.

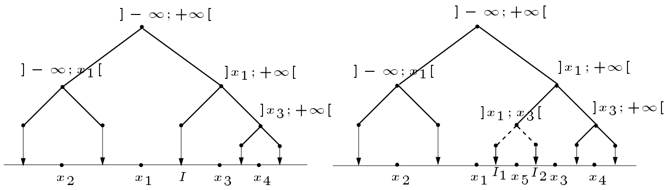


Figure 1: *Inserting x_5 in the binary tree*

The nodes of this binary tree are intervals, the two sons of a node correspond to the splitting of that interval into two sub-intervals. Thus, when a new number is inserted, it is located in the binary tree, the leaf containing it becomes an internal node (I in the example Fig.1), and its interval is split into two with respect to the newly inserted number (I_1 and I_2). This scheme is exactly equivalent to quicksort if the pivot chosen is the first element of the array.

This is clearly an incremental algorithm, updating a result each time a new object is inserted. Obviously, the time complexity of this algorithm depends on the actual input order.

2.2 Performance of the algorithm

In this section, we investigate relative performances of the algorithm depending on the input order.

Worst-case analysis We start by considering a deterministic version of this algorithm i.e., by assuming that the data are provided in a fixed order defined by the user.

Theorem 2.1 *The worst-case complexity of the sorting algorithm is* $\Theta(n^2)$.

Proof: For the particular version we have chosen, the worst case occurs when the input is already sorted. Inserting the numbers in that order produces a degenerate binary tree looking like a comb.

In this case, the location of each newly inserted number in the binary tree needs to compare this number with the set of all numbers already present in the binary tree.

Total sorting time is then obtained by summing this cost over the whole sequence of insertion i.e. $\sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Average case analysis Looking into the average behaviour of the algorithm, we introduce a simple assumption in the analysis: all $n!$ possible orders to introduce the n data are equally likely.

Theorem 2.2 *The average case complexity of the sorting algorithm is* $\Theta(n \log n)$.

Proof: Let us use backward analysis [13] to estimate the total expected location cost of the algorithm. To calculate the expected cost of the $(i+1)^{st}$ insertion, we *imagine*, just for the sake of analysis, going backwards from the $(i+1)^{st}$ to the i^{th} step.

Let T_j denote the search tree right after insertion of x_j. Just after the insertion of the j-th number $j < i$, a single leaf $[a, b]$ of T_j contains x_i, this leaf is created at the j-th step if either a or b is inserted at this step. Moreover, there exists exactly $j+1$ leaves in T_j. Thus, we have "x_i belongs to an interval created at step $j < i$ with probability $\frac{2}{j+1}$". The expected cost of inserting the i-th number is then obtained by summing over the step of creation of these intervals i.e.

$$E[cost(x_i)] = \sum_{j=1}^{i-1} \frac{2}{j+1} = 2(H_i - 1) \leq 2 \log i$$

where H_i denotes the i^{th} harmonic number and is defined as $H_i = \sum_{j=1}^{i} \frac{1}{j}$.

The overall expected sorting cost of the n numbers is then

$$\sum_{i=1}^{n} 2 \log i \leq 2n \log n = \Theta(n \log n)$$

Looking into the average behaviour of the algorithm, we have introduced a simple assumption in the analysis: all $n!$ permutations are equally likely.

When this assumption about the input distribution is valid, the algorithm has an optimal $\Theta(n \log n)$ expected complexity. Thus, an absolute guarantee of a fast average running time is guaranteed when combining the algorithm with a random permutation of the objects to be processed. It is important to note that this modification does not improve the worst case behaviour but the expected running time of the algorithm is now independent of input ordering. Random permutation protects against the occurence of the worst case that becomes extremely unlikely since the algorithm has a bad behaviour only if the random choices elicit an unfortunate permutation.

However, in practice, if the data are known in advance, the random order of the points can be ensured by shuffling them but the dynamic aspect of the algorithm is lost. Thus the question arises, whether it is still possible to use this kind of randomization technique for an on-line algorithm i.e. if the numbers are no longer random but chosen by an adversary.

***Shuffling buffer* analysis** This section introduces the shuffling buffer model studied afterwards. It consists of an array of size k plugged in between the process providing the input data and the algorithm. It allows us to locally shuffle the initial adversary order, and therefore, to introduce some randomness without having to wait for all the data.

Now, let us describe a possible strategy using this model. The k first numbers given by the adversary fill the buffer. They are then passed to the algorithm by emptying out the buffer in a random order which is equivalent to choosing a random permutation among $k!$ and can be done in linear time. The procedure is then repeated by filling the buffer with the next k numbers given by the adversary. This technique permits to generate $(k!)^{\lfloor \frac{n}{k} \rfloor}(n - \lceil \frac{n}{k} \rceil k)!$ different insertion orders for the underlying algorithm.

Clearly, if $k = n$ (i.e. if the buffer is large enough to contain all the input data), we have a random order, and if $k = 1$, the adversary order is unchanged. We investigate intermediate situations with $1 < k << n$.

Theorem 2.3 *The expected complexity of the sorting algorithm using the shuffling buffer is* $\Theta(\frac{n^2 \log k}{k})$.

Proof: We are interested in the maximum expected running time of the algorithm, where the maximum is taken over all input orders (i.e., worst adversary order), and the expectation is taken over all possible permutations of the buffers. Obviously, for the first k inserted numbers, we obtain the cost of the randomized analysis i.e., a balanced tree on average.

Given x_p, the p^{th} inserted number, $p > k$, we want to count the number of comparisons of x_p with numbers of the first buffer $\{x_i, i \leq k\}$. The numbers of the first buffer can be renamed $b_1 < b_2 < \ldots < b_k$ in their sorted order, and let j be such that $b_j < x_p < b_{j+1}$. The probability that x_p is compared with b_i $(i \leq j)$ is equal to the probability that b_i is inserted before the $j - i$ numbers lying between b_i and x_p i.e., $\frac{1}{j-i+1}$. In the same way, the probability that x_p is compared with b_s $(j < s \leq k)$ is equal to the probability that b_s is inserted before the $s - (j + 1)$ numbers lying between x_p and b_s i.e., $\frac{1}{s-j}$.

The expected insertion cost of the p^{th} number with respect to j is then given by the following equation:

$$\sum_{i=1}^{j} \frac{1}{j-i+1} + \sum_{s=j+1}^{k} \frac{1}{s-j} = H_j + H_{k-j}$$

where $H_i = \sum_{j=1}^{i} \frac{1}{j}$ denote the i^{th} harmonic number. The function $f(j) = H_j + H_{k-j}$ is concave on the interval $0, \ldots, k$, and reaches its maximum for $j = \lfloor \frac{k}{2} \rfloor$. Thus, the interval defined by the $\lfloor \frac{k}{2} \rfloor^{th}$ and the $(\lfloor \frac{k}{2} \rfloor + 1)^{st}$ numbers of the sorted list of the k elements is the deepest leaf on average of the randomly constructed binary tree.

One can easily prove that, in the worst case, each of the k elements of the next buffer is precisely located in this leaf. Each set of k insertions then creates a sub-tree rooted at this particular leaf of the previous constructed sub-tree. Such an order occurs when the interval defined by two consecutive numbers contains all the numbers provided afterwards by the adversary. This order is, of course, a worst case of the deterministic version of the algorithm similar to the sorted order.

A bound on the expected cost of the p^{th} insertion in the tree is then obtained by summing the expected maximal localization cost over the $\lfloor \frac{p-1}{k} \rfloor$ already constructed subtrees and the subtree in the course of construction. With $H(k) = H_{\lfloor \frac{k}{2} \rfloor} + H_{\lceil \frac{k}{2} \rceil} = O(\log k)$, we obtain:

$$E[cost(x_p)] \leq \left(\left\lfloor \frac{p-1}{k} \right\rfloor + 1 \right) H(k) = O(\frac{p}{k} \log k)$$

The overall expected sorting time arises when summing over the set of n numbers, i.e.,

$$E[Overallcost] = \sum_{p=1}^{n} E[cost(x_p)] = O(\frac{n^2 \log k}{k})$$

For small values of k in comparison with n, overall expected sorting time is quadratic, nevertheless, we have improved the worst case complexity with a $\frac{H(k)}{k}$ factor. When the buffer size tends towards n, the total data number, we find again the $O(n \log n)$ complexity of the randomized case.

3 Delaunay Triangulation

Computing the Delaunay triangulation of a set of points is a classical problem in computational geometry, and it constitutes one of the most popular structures in this field due to its numerous applications (terrain models, 3D reconstruction, meshes ...).

Let $\mathcal{S}$ be a set of n points in the plane. The Delaunay triangulation of $\mathcal{S}$, for short $DT(\mathcal{S})$ is a plane graph whose vertices are the points in $\mathcal{S}$. It is usually defined as the unique triangulation such as the circles circumscribing the Delaunay triangles do not contain any site in their interior. This property allows us to design very simple incremental algorithms that progressively acquire new data, which is of great practical interest.

For the sake of ease of presentation let us assume we are dealing with a set $\mathcal{S}$ of n points in non-degenerate (or general) position, which in this case means that no three points are collinear and no four points are cocircular.

3.1 Description of the algorithms

They consist of inserting the points in turn, and in updating the structure after each insertion. When a new site M is introduced in the triangulation, two steps are performed: firstly, we locate M in order to find and remove the set of all the triangles whose circumscribing circle contains M (*location step*); secondly, we create the new triangles (*insertion step*) by linking M to the elements of the set $F(M)$ of edges of the boundary of the simply connected region $R(M)$, the union of the triangles in conflict with M (see Fig.2).

For practical reasons, an additional point is added at infinity in order to triangulate the unbounded face too. Thus, the convex hull boundary case is processed in an easier way. All the vertices belonging to the convex hull are incident to this point, and an infinite triangle is associated to all the edges of the convex hull.

Based on this approach, there exists several incremental algorithms. They differ mainly in the way they find $R(M)$ [2, 12, 7, 8, 9, 11].

The complexity of the location step, which strongly depends on the location strategy will be evoked in the next section. We first focus our interest on the triangulation of $R(M)$.

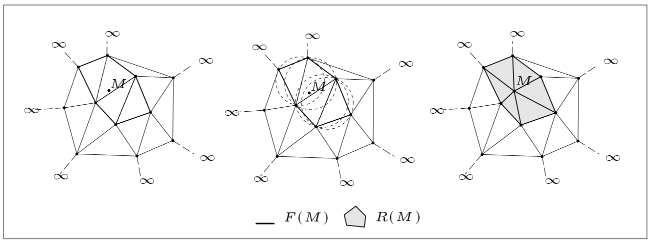


Figure 2: *Inserting a site in the Delaunay triangulation.*

3.2 Insertion step analysis

We study here the cost of retriangulating $R(M)$, which is independent of the location technique, and common to all incremental algorithms.

The number of constructed triangles during the insertion of M in the structure is exactly equal to the degree of M in the resulting triangulation (see Fig.2).

Worst-case analysis

Theorem 3.1 *The worst-case number of triangles created by an incremental Delaunay construction is* $\Theta(n^2)$.

Proof: In the worst case (points lying on a piece of parabola and provided precisely in the direction of the increasing curvature) the degree of the last inserted point is linear in the number of points already inserted in the triangulation. The incremental process then needs the creation of $\sum_{i=3}^{n}(i-2) = \frac{(n-2)(n-1)}{2} = \Theta(n^2)$ triangles.

Randomized analysis A well known result about the incremental construction of a Delaunay triangulation is that randomization leads to constant size $R(M)$ regions on average, regardless of the number of points, and thus it minimizes the number of triangles that must be constructed.

Theorem 3.2 *The average-case number of triangles created by an incremental Delaunay construction is* $\Theta(n)$.

Proof: If M is chosen from $\mathcal{S}$ uniformly at random, then the expected degree of M in $DT(\mathcal{S})$ is the degree of a random point in the Delaunay triangulation, which is well known to be at most equal to 6 [13].

Hence, the overall expected number of triangles created is bounded by $\sum_{i=1}^{n} 6 = \Theta(n)$.

Shuffling buffer analysis How does this relate to our shuffling buffer technique?

Let $\mathcal{V}$ be a set of $m - j$ sites for which the triangulation is already constructed, and $\mathcal{W}$ be another set of j sites. The points of $\mathcal{W}$ are introduced one after the other in a random sequence updating the Delaunay triangulation of the current set until construction of $DT(\mathcal{V}\cup\mathcal{W})$. In this case, the last inserted point is no more chosen uniformly at random among the m sites belonging to $\mathcal{V}\cup\mathcal{W}$, as assumed in a randomized analysis, but among the points belonging to $\mathcal{W}$, i.e., the j points that are in the last buffer.

Theorem 3.3 *The expected number of triangles created by an incremental Delaunay construction using the shuffling buffer is* $\Theta(\frac{n^2 \log k}{k})$.

Proof: Let us express as in *backwards analysis* the cost of the "last step" as a function of the produced output. M being chosen uniformly at random from $\mathcal{W}$, the average degree of x_m in $DT(\mathcal{V}\cup\mathcal{W})$ equals the sum of the degrees over the j sites of $\mathcal{W}$ divided by j. If $E[cost(x_m)]$ denotes the insertion part of the expected cost of inserting x_m, we obtain:

$$\begin{aligned} E[cost(x_m)] &= \frac{1}{j}\sum_{s\in\mathcal{W}} \text{degree}(s, DT(\mathcal{V}\cup\mathcal{W})) \\ &\leq \frac{1}{j}\sum_{s\in\mathcal{V}\cup\mathcal{W}} \text{degree}(s, DT(\mathcal{V}\cup\mathcal{W})) \end{aligned}$$

and as the average degree of s in $DT(\mathcal{V}\cup\mathcal{W})$ is at most 6,

$$E[cost(x_m)] \leq \frac{1}{j} 6|DT(\mathcal{V}\cup\mathcal{W})| = \frac{6m}{j}$$

This is the cost of the m^{th} point which is the j^{th} in its buffer. Thus, the cost of the $(q+1)^{st}$ buffer is obtained with $m = qk + j$ and summing over j. The overall expected insertion cost of the Delaunay triangulation of n points is then less or equal to

$$\sum_{q=0}^{\lceil\frac{n-k}{k}\rceil}\sum_{j=1}^{k}\frac{6(qk+j)}{j} \leq \sum_{q=0}^{\lceil\frac{n-k}{k}\rceil} 6(qkH_k + k) = \Theta\left(\frac{n^2\log k}{k}\right)$$

Tightness For $k = \Theta(n)$, the bound seems to be worse than the $O(n)$ randomized bound, but actually the $\log k$ factor is tight if there is more than one buffer. An explicit construction (the parabola example which was introduced as the worst case of our incremental deterministic algorithm) shows that $k = \frac{n}{2}$ leaves an adversary enough freedom for *bad* inputs taking time $\Omega(n \log n)$ (see Figure 3).

It suffices to consider the cost of insertion of the second buffer to prove optimality of Theorem 3.3 since the cost of inserting the p^{th} point x_p $(p > k)$ can be analyzed as follows:

- x_p has minimal abscissa with probability $\frac{1}{p-k}$, its insertion cost is then $p-2$.
- x_p has non minimal abscissa with probability $\frac{p-k-1}{p-k}$, its insertion cost is then 2.

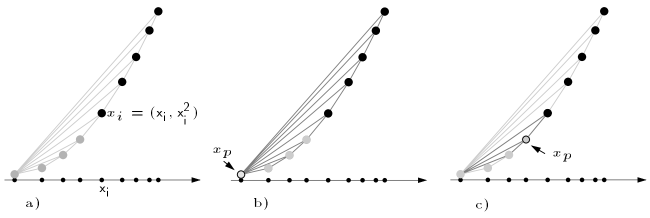


Figure 3: *a) The Delaunay triangulation of the parabola example with $n = 2k$, the points of the first buffer are black, the points of the second buffer are grey. b) The last inserted point x_p is minimal. c) The last inserted point x_p is non minimal.*

which leads to

$$
\begin{aligned}
E[cost(2^{nd} \text{ buffer})] &= \sum_{p=k+1}^{2k} \left(\frac{1}{p-k}(p-2) + \frac{p-k-1}{p-k} 2 \right) \\
&= \sum_{j=1}^{k} \frac{3j-4+k}{j} = \Omega(k \log k)
\end{aligned}
$$

Thus, we get a total cost $E[Overallcost] \geq \Omega(k \log k)$ which proves that $n^2 \frac{\log k}{k}$ is asymptotically optimal.

3.3 Location step analysis

In this abstract, we have focussed our attention on the analysis of structural changes in the Delaunay triangulation.

The analysis of the structural changes in the triangulation is a lower bound for the time complexity of a Delaunay incremental construction, however, the shuffling buffer strategy has also an influence on the main part of the cost that comes from the point location strategy. We present below some results.

The straightforward strategy which examines all the triangles is quadratic in any cases and is not affected by the order. Location using *walk* is quadratic without hypothesis on the input distribution, and this bound is not improved by a random order hypothesis.

The *jump and walk* strategy [11] locates a new point by searching its nearest neighbor in a sample of m points and walking from there to find $R(M)$. Choosing $m = \sqrt{n}$ and $k = \sqrt{n} \log n$ gives the same complexity $O(n\sqrt{n})$ in the randomized and shuffling buffer analysis, this is the best randomized complexity without hypotheses on the point distribution.

Several randomized incremental construction of the Delaunay triangulation such as the Delaunay tree [2] use a graph storing the history of the triangulation construction [7, 8], then the location phase finds all the triangles that have existed during the construction and whose circumcircle contains the located point; this yields to a (disastrous) $O(n^3)$ complexity which is improved in $O(\frac{n^3 \log k}{k})$ by the shuffling buffer.

Some other location techniques based on a hierarchy of random samples independent from the insertion order (similarly to skip-lists) such as the Delaunay hierarchy [4] gives an expected linear location time for a non random order. The shuffling buffer technique improves it by the $\frac{\log k}{k}$ factor getting a location time of the same order as the insertion time. The whole complexity of constructing the Delaunay triangulation of points in the plane with the Delaunay hierarchy and the shuffling buffer is $O(\frac{n^2 \log k}{k})$.

3.4 Experimental results

We performed experimental testing on 100000 2D points lying on a piece of parabola and ordered in the direction of the increasing curvature. The results confirm the previous theoretical analysis (see Figure 4).

We also present experiments on real 3D data provided by a scanning system measuring real objects having respectively 145300 and 407500 3D-points (see Figure 5). Such a measuring system really matches the context of that paper, data are measured by moving a scanning system around the object and thus data are not all known in advance and they are not coming in a random order.

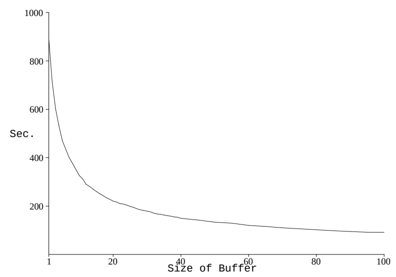


Figure 4: *Running time of the Delaunay triangulation. Points on parabola.*

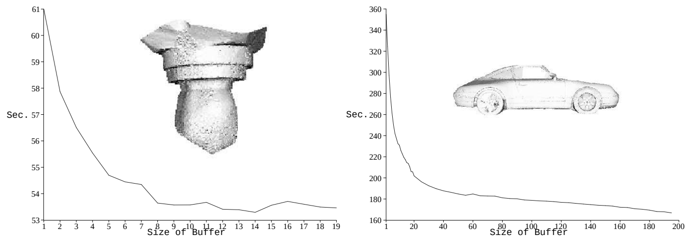


Figure 5: *Running time of the Delaunay tetrahedrization. Courtesy of Kreon Industry.*

We use an implementation of the Delaunay Hierarchy [4]. As Figure 5 testifies, experimental results, obtained by averaging on about 50 different executions, indicate that the shuffling buffer version is efficient in practice.

4 Trapezoidal diagrams and line segment intersections

For $\mathcal{S}$ a set of n straight line segments in the plane, what are the pairs of intersecting segments in $\mathcal{S}$?

This computational problem has received much attention since the need to calculate efficiently the intersection points of sets of line segments arises naturally from the manipulation of display and related entities through, for instance, windowing, clipping, and hidden surface removal.

In this section, we are interested in finding the trapezoidal diagram of $\mathcal{S}$ where $\mathcal{S}$ is a set of n straight line segments in the plane. As a byproduct, all intersecting pairs of segments of $\mathcal{S}$ can be found.

For the sake of ease of presentation we again assume that we are dealing with a set $\mathcal{S}$ of n segments that is non-degenerate in the sense that no two segments of $\mathcal{S}$ have the same endpoint, no three segments intersect in a common point, no two segment endpoints have the same x-coordinate, and that no segment endpoint lies in the relative interior of some other segment.

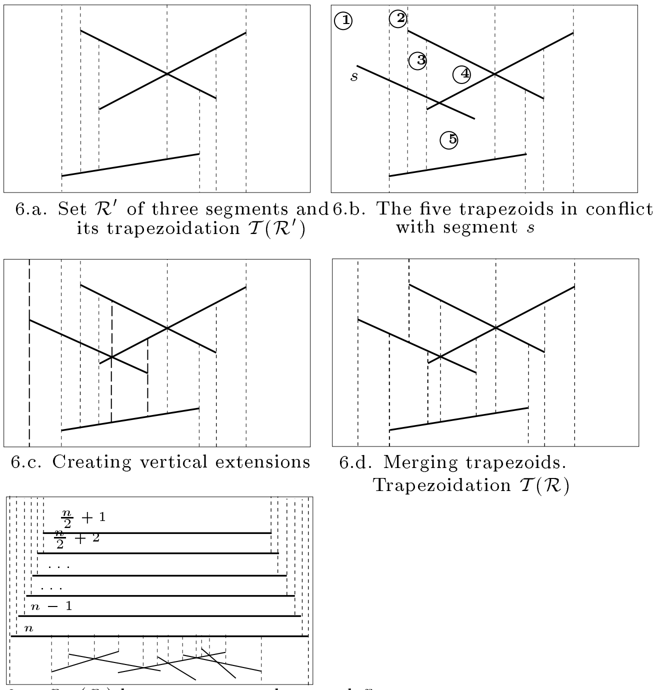
4.1 Description of the algorithm

The trapezoidal diagram (or "vertical visibility map"), denoted $\mathcal{T}(\mathcal{S})$, is defined as follows: for every point p that is either an endpoint of a segment in $\mathcal{S}$, or an intersection point of two segments in $\mathcal{S}$, extend a vertical segment from p to the first segment of $\mathcal{S}$ above p, and to the first segment of $\mathcal{S}$ below p. If no such segment is "visible" to p above it, then extend a vertical ray above p, and similarly below. The resulting vertical segments, together with the segments in $\mathcal{S}$, form a subdivision of the plane into simple regions (triangles or trapezoids) called the trapezoidal diagram.

We now detail how a segment s is introduced into the current trapezoidal map $T(\mathcal{R}')$ to obtain $T(\mathcal{R})$, where $\mathcal{R}' = \mathcal{R} \setminus \{s\}$ (see Figure 6.a-d).

As for the Delaunay triangulation, two different operations need to be addressed in the incremental construction of line segment arrangements. One is the *point location* of (typically the left) endpoint of a new segment s in the vertical trapezoidal decomposition of the arrangement of the segments inserted so far. The other is the *propagation* or *insertion* of s through the trapezoidal decomposition, in order to discover the intersections of s with existing segments, and to update the trapezoidal decomposition in the process.

This update creates the new trapezoids of $T(\mathcal{R})$. This involves splitting the trapezoids of $T(\mathcal{R}')$ that are intersected by s (five trapezoids in our example Figure 6.b), introducing the vertical extensions from the endpoints of s and the intersection points of s with the other segments of $\mathcal{R}$ (see Figure 6.c), and merging



6.a. Set $\mathcal{R}'$ of three segments and its trapezoidation $\mathcal{T}(\mathcal{R}')$

6.b. The five trapezoids in conflict with segment s

6.c. Creating vertical extensions

6.d. Merging trapezoids. Trapezoidation $\mathcal{T}(\mathcal{R})$

6.e. $\frac{n}{2}\alpha(\frac{n}{2})$-large upper envelope and $\frac{n}{2}$ parallel segments inserted top to bottom.

Figure 6: *Inserting a line segment in trapezoidal map*

trapezoids that are separated by vertical edges that are not part of vertical extensions any more because of the introduction of s (see Figure 4.d).

4.2 Theoretical analysis

The point location step strongly depends on the location strategy [12, 6] and the analysis is focused on the cost of the insertion step, that is clearly proportional to the number of trapezoids appearing in $T(\mathcal{R})$ which did not exist in $T(\mathcal{R}')$. For this particular problem, the output size a, namely the number of intersecting pairs of segments in $\mathcal{S}$, is needed as a complexity parameter.

When a new segment s is inserted, it creates itself four new trapezoids (one face having s as floor, one having s as ceil, and one face for each of the two new vertical extensions introduced from the endpoints of s). For each intersection points of s with the other segments of $\mathcal{R}'$, one adds four others trapezoids (two trapezoids for each vertical extensions introduced from the intersecting point and two trapezoids having a single vertical partition). At last, one creates one new face for each of the vertical extensions intersected by s.

Thus, the number of trapezoids constructed at the time of insertion of s in $T(\mathcal{R}')$ is:

$$4 + 4a_s + l_s$$

where a_s denotes the number of intersection points of s with the other segments of $\mathcal{R}'$, and l_s denotes the number of vertical extensions of $T(\mathcal{R}')$ intersected by s.

Randomized analysis

Theorem 4.1 *The average number of trapezoids created by an incremental trapezoidal map construction is $\Theta(n+a)$.*

Proof: Assume in a first time, that the n segments are inserted at random and let us apply backwards analysis. The important step is to express the cost of introducing a segment s into $T(\mathcal{R}')$ in terms of the resulting graph $T(\mathcal{R})$ and not in terms of $T(\mathcal{R}')$.

It is not hard to see that if s is chosen at random among the r segments of $\mathcal{R}$, the insertion cost of the last segment is given by

$$\frac{1}{r}\sum_{s\in\mathcal{R}}(4+4a_s+l_s) = 4+4\frac{2a_{\mathcal{R}}}{r}+\sum_{s\in\mathcal{R}}\frac{l_s}{r},$$

since each segment intersection is counted twice in the sum.

We still have to determine the expected value of $a_{\mathcal{R}}$, the number of intersecting pairs of segments in a random sample $\mathcal{R}$, that is a subset $\mathcal{R}\subseteq\mathcal{S}$ of size r chosen at random.

If $\mathcal{R}$ is a r-sample of $\mathcal{S}$, the expected number of $a_{\mathcal{R}}$ is

$$\frac{ar(r-1)}{n(n-1)}.$$

Indeed, an intersection point between two segments of $\mathcal{S}$ is an intersection point of two segments of $\mathcal{R}$ if the two segments of $\mathcal{S}$ that intersect belong to $\mathcal{R}$, which happens with probability $\frac{\binom{n-2}{r-2}}{\binom{n}{r}} = \frac{r(r-1)}{n(n-1)}$

Moreover, the trapezoidal diagram of a r-sample $\mathcal{R}$ has exactly $2r+a_{\mathcal{R}}$ vertical segments, since each one of these is at most intersected two times by segments belonging to $\mathcal{R}$, we have:

$$\frac{1}{r}\sum_{s\in\mathcal{R}} l_s \le \frac{2}{r}(2r+a_{\mathcal{R}}) = 4+\frac{2a(r-1)}{n(n-1)}$$

Thus the expected cost of introducing the r^{th} segment into $T(\mathcal{R}')$ is bounded by

$$4+4\frac{2a(r-1)}{n(n-1)}+4+\frac{2a(r-1)}{n(n-1)} = 8+10\frac{a(r-1)}{n(n-1)}.$$

To obtain an upper bound of the expected cost for all insertions of the entire algorithm, one clearly only needs to sum this expression for $1\le r\le n$, which yields

$$E[Overall cost] \le 8n+5a = \Theta(n+a)$$

Worst-case analysis

However, one cannot suppose that the segments are inserted at random because of the on-line constraint. Thus let us assume now that the segments are inserted in an order chosen by an adversary and let us apply a classical worst case analysis.

Theorem 4.2 *The worst-case number of trapezoids created by an incremental trapezoidal map construction is $\Theta(n^2\alpha(n))$.*

Proof: Just before the $(i+1)^{st}$ insertion, there exists exactly $2i+a_{\mathcal{S}_i}$ vertical extensions in the current trapezoidal map $\mathcal{T}(\mathcal{S}_i)$. Moreover, it is well known [14] that the complexity of the lower envelope of a set of n line segments depends almost linearly on the number of segments and it is $\Theta(n\alpha(n))$, where $\alpha(n)$ denotes the functional inverse of the *Ackerman function*, which is an extremely slow-growing function.

Thus, in the worst case, the cost of introducing the $(i+1)^{st}$ segment in the current trapezoidal map $\mathcal{T}(\mathcal{S}_i)$ is

$$cost_{x_{i+1}} = 4+4a_{x_{i+1}}+2i+\min(i\alpha(i),a_{\mathcal{S}_i}).$$

Again, an upper bound on the cost for all insertions of the entire algorithm simply arises when summing this expression over the n insertions which yields:

$$\sum_{i=0}^{n-1} cost_{x_{i+1}} = n(n-1)+4n+4a+\sum_{i=0}^{n-1}\min(i\alpha(i),a_{\mathcal{S}_i})$$

$$\le n^2+3n+4a+\min(\frac{n^2}{2}\alpha(n),an) = O(n^2\alpha(n))$$

Figure 6.e shows an example of complexity $\Omega(n^2\alpha(n))$, thus the bound is tight.

Shuffling buffer analysis

Theorem 4.3 *The expected number of trapezoids created by an incremental trapezoidal map construction using the shuffling buffer is $\Theta(n^2\alpha(n)\frac{\log k}{k})$.*

Proof: For the shuffling buffer version, the last inserted segment is chosen uniformly at random among the k segments belonging to the last buffer $\mathcal{W}$. Hence, the expected insertion cost of the x_m, the last segment in the p^{th} buffer $\mathcal{W}$ is given by

$$\mathrm{E}[\mathrm{cost}(x_m)] = \frac{1}{k}\sum_{s\in\mathcal{W}}(4+4a_s+l_s) \le 4+\frac{4}{k}a_p+\frac{1}{k}\sum_{s\in\mathcal{S}} l_s$$

$$\le 4+\frac{4}{k}a_p+\frac{4m+2\min(a,m\alpha(m))}{k},$$

where a_p denotes the number of intersections of segments in p^{th} buffer with any other segment.

Like the previous shuffling buffer analysis, we repeatedly delete random segments of the current buffer. An upper bound on the cost of insertion of the $(p+1)^{st}$ buffer is then:

$$\sum_{i=1}^{k}\left(4+\frac{4}{k}a_p+\frac{4(pk+i)+2\min(a,n\alpha(n))}{i}\right)$$

$$\leq 4a_p+4pkH_k+8k+2\min(a,n\alpha(n))H_k.$$

Using the fact that $\sum_{p=0}^{\lfloor\frac{n}{k}\rfloor}a_p\leq 2a$, and summing over the buffers to be processed, we obtain the following upper bound on the overall expected insertion cost:

$$\sum_{p=0}^{\lfloor\frac{n}{k}\rfloor}(8k+4a_p+4pkH_k+2\min(a,n\alpha(n))H_k)$$

$$\leq 2\min(a,n\alpha(n))H_k(\frac{n}{k}+1)+4\frac{H_k}{k}n^2+\Theta(a+n)$$

$$=\Theta(n^2\alpha(n)\frac{\log k}{k})$$

Using the buffer model, the expected number of constructed trapezoids during the incremental process is then almost quadratic with respect to the number of segments in the worst-case. Nonetheless, we have reduced the constant of the quadratic term obtained in the worst-case analysis by a $\frac{\log k}{k}$ factor.

As for result the Delaunay triangulation taking $k=\Omega(n)$ and $a=O(n)$ yields to a $O(n\alpha(n)\log n)$ bound which is almost tight if there is more than one buffer (see Figure 6.e for $k=\frac{n}{2}$).

For an algorithm constructing the map, we have to use a location technique to find where to insert a new segment. As for Delaunay triangulation, a hierarchy of random samples independent from the insertion order gives good results. This hierarchy can be maintained dynamically: when a new segment is inserted, it is decided at random to which samples it belongs.

5 Conclusion

We have shown that the shuffling buffer permits to improve the incremental construction cost of structures such as the binary search tree, the Delaunay triangulation or trapezoidal maps. For all studied algorithms, a significant improvement of a $O(\frac{\log k}{k})$ factor on their worst case complexities has been proved. In an incremental Delaunay algorithm, the number of constructed triangles goes from $O(n^2)$ to $O(\frac{n^2\log k}{k})$. This analysis of the structural changes in the triangulation is a lower bound for the time complexity of a Delaunay incremental construction, however, the main part of the cost comes from the point location strategy. We gave results for several strategies in this paper, in the case of the Delaunay hierarchy [4], we achieve optimal randomized complexity of $O(n^2)$ in the worst case and $O(\frac{n^2\log k}{k})$ with the shuffling buffer technique. Similarly the number of trapezoids created in an incremental trapezoidal map construction goes from $O(n^2\alpha(n))$ to $O(\frac{n^2\alpha(n)\log k}{k})$.

The $\frac{\log k}{k}$ factor of improvement has been proven in the worst case, that is for the worst configuration of the data given in the worst order. In practice, however, the order of the data is neither randomized nor the worst case. Experimental results performed for the planar case and on real data sets provided by a 3D scanning system indicate that the shuffling buffer principle permits a significative improvement of the running time of the Delaunay algorithm in practice.

The shuffling buffer principle coming in between the process providing the data and the input of the algorithm has the advantage of not modifying the underlying algorithm. Nevertheless, if the data order is completely fixed, it may be better to take into account the knowledge of this order to minimize the complexity of the algorithm.

We have studied a second strategy that consists of replacing at each step the object chosen by the new object given by the adversary instead of emptying the buffer. It turns out that this strategy which is more complex to analyze is moreover less efficient. This can be proved theoretically for the sorting algorithm when $k>7$ and confirmed experimentally for the incremental construction of Delaunay triangulation [5].

References

[1] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete Comput. Geom.*, 8:51–71, 1992.

[2] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. Technical Report 1140, INRIA Sophia-Antipolis, Valbonne, France, 1989.

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

[4] O. Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[5] O. Devillers and P. Guigue. Le tampon mélangeur. Rapport de recherche 3988, INRIA, 2000.

[6] O. Devillers, M. Teillaud, and M. Yvinec. Dynamic location in an arrangement of line segments in the plane. Rapport de recherche 1558, INRIA, 1991.

[7] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[8] R. Klein, K. Mehlhorn, and S. Meiser. Randomized incremental construction of abstract Voronoi diagrams. *Comput. Geom. Theory Appl.*, 3(3):157–184, 1993.

[9] J.-M. Moreau. Hierarchical Delaunay triangulation. In *Proc. 6th Canad. Conf. Comput. Geom.*, pages 165–170, 1994.

[10] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, New York, NY, 1995.

[11] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

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

[13] R. Seidel. Backwards analysis of randomized geometric algorithms. Report TR-92-014, Computer Science Division, University of California, Berkeley, Feb. 1992.

[14] A. Wiernik and M. Sharir. Planar realizations of nonlinear Davenport-Schinzel sequences by segments. *Discrete Comput. Geom.*, 3:15–47, 1988.