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 [11, 9, 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 kind of randomized technique is suitable in many domains and permits to introduce enough randomness to guarantee some improvement on the worst case complexity without modifying the underlying algorithm.

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). 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 provides experimental results for 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.

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.

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, and its interval is split into two with respect to the newly inserted number. 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: In the worst case the input is already sorted. In this case the i^{th} insertion costs $i-1$ comparisons. This gives a worst-case time estimate of $\sum_{i=1}^{n} i-1 = \frac{n(n-1)}{2} = \Theta(n^2)$.

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: If the i^{th} inserted number is chosen from the n number input uniformly at random, then a *backward analysis* [12] gives that its average insertion cost is $\Theta(\log i)$. The overall expected sorting cost is then $\sum_{i=1}^{n} \Theta(\log i) = \Theta(n \log n)$.

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.

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

Let $\mathcal{S}$ be a set of n points in the plane in *non-degenerate* position. 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.

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.1).
Based on this approach, there exists several incremental algorithms. They differ mainly in the way they find $R(M)$ [2, 11, 6, 7, 8, 10].

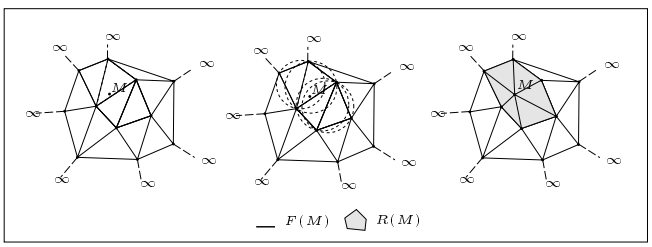

Figure 1: *Inserting a site in the Delaunay triangulation.*

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.1).

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

Shuffling buffer analysis 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 (points lying on a curve with increasing curvature and provided precisely in the direction of the increasing curvature) shows that $k = \frac{n}{2}$ leaves an adversary enough freedom for *bad* inputs taking time $\Omega(n \log n)$; hence $n^2 \frac{\log k}{k}$ is asymptotically optimal.

3.3 Location step analysis

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 (without the detailed analysis by lack of space). The *jump and walk* strategy [10] 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. 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.

3.4 Experimental results

We performed experimental testing on real data provided by a 3D scanning system measuring real objects having respectively 145300 and 407500 3D-points.

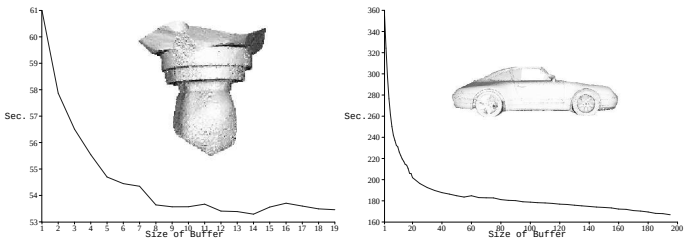

Figure 2: *Running time of the Delaunay tetrahedrization. Courtesy of Kreon Industry.*

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. We use an implementation of the Delaunay Hierarchy [4]. As Figure 2 testifies, experimental results, obtained by averaging on about 50 different executions, indicate that the shuffling buffer version is efficient in practice.

4 Conclusion

We have shown that the shuffling buffer permits to improve the incremental construction cost of structures such as unbalanced binary search tree or Delaunay triangulation. For both 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 $O(\frac{n^2 \log k}{k})$ upon use of a shuffling buffer of size k. Similarly the number of trapezoids created in an incremental trapezoidal map construction can be proved [5] to go from $O(n^2\alpha(n))$ to $O(\frac{n^2\alpha(n) \log k}{k})$ where α is the inverse Ackerman function.

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 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.

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

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

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

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

[10] 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.

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

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