Introduction to Randomized Algorithms

Chee Yap

April 21, 2003

1 Introduction

Randomness is a powerful tool in algorithms. The most well-known example is Quicksort invented by Tony Hoare. In number-theoretic algorithms such techniques are also very important. In the 1990's it was realized that randomness can be exploited very naturally in many geometric algorithms. Clarkson and Mulmuley were among the first developers of such techniques. These random algorithms are typically simple to implement and relatively straightforward: such ought to be the algorithms of choice for implementors. It turns out that there are two general paradigms that capture many randomization approaches:
(1) randomized divide and conquer and
(2) randomized incremental construction. Both will viewpoints will be discussed.

2 Examples of Geometric Computation

We shall introduce randomized algorithms via several examples of in computational geometry.

The first is the sorting of numbers, something that every beginning student of computer science knows. While this may not strike you as geometry, it is really one-dimensional geometry. Given a set S of n numbers, we want to sort them. There are standard algorithms to compute this is $O(n \log n)$. We are interested in a particular method, Quicksort. Idea is to randomly pick a point $x \in S$, and split the problem into two halves; recurse on each half.

Trapezoidal Decomposition: given a set S of n line segments in the plane, compute its trapezoidal decomposition. Say there are k pairwise intersections among the segments, $0 \leq k \leq \binom{n}{2}$. This defines $O(n+k)$ trapezoids.

FIGURE 1.

This means we want to compute the set of trapezoids and their adjacency relationship in $O(n \log n + k)$ time. Chazelle has shown that this bound can be achieved deterministically but it is not a practical algorithm. Our idea is to randomly insert successive one segment at a time.

Convex Hull: given a set of points, compute its convex hull. This problem might be said to be the simplest non-trivial geometric problem. In 1-dimensions, it corresponds to finding the maximum and minumum of a set of numbers. As we said before, in computer graphics, you need it in some Bezier curve algorithms.

Linear Programming: given a set of half spaces $h_1, \ldots, h_n$, we want to compute a point x in their intersection $\cap_{i=1}^n h_i$ so as to minimize a linear function $f(x)$. This problem is very important in many practical applications such as arise in economics.

3 Quick Probability

Since we want to do randomization, we should review our basic facts of probability. We do not need much at all, in fact, I intend to give you all that you need here.

In any probabilistic analysis, you must look for the underlying probability space. This is often implicit. But in its simplest form (that is all we need here), it is extremely simple: you need to look for a set Ω and a function $\Pr : \Omega \to [0,1] = \{x : 0 \le x \le 1\}$. A probability space is just this pair, $(\Omega, \Pr)$.

The set Ω is called the **sample space**, and we take it to be any finite set. Subsets $A \subseteq \Omega$ are called **events**. An element $\omega \in \Omega$ is called a **sample point**, and we do not distinguished it from the corresponding event $\{\omega\}$.

The function $\Pr : \Omega \to [0,1]$ is the **probability distribution function**. It extends to events $A \subseteq \Omega$ as follows: $\Pr(A) = \sum_{\omega \in A} \Pr(\omega)$ This gives us two properties:
(A1) $\Pr(\Omega) = 1$
(A2) $A, B \subseteq \Omega$ implies $\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(B \cap A)$.
Alternatively, in the general setting where A may not be finite, we can use (A1, A2) as axioms.

Random Variables. Let $\mathbb{R}$ be the real numbers. A **random variable** (r.v.) is any function

$$X : \Omega \to \mathbb{R}.$$

If X_i are r.v.'s, then so are

$$X_1 \pm X_2, \quad X_1 X_2, \quad X_1^{X_2}, \ldots$$

The **expection** of a r.v. X is given by

$$E[X] := \sum_{\omega \in \Omega} X(\omega) \Pr(\omega)$$

A remarkable elementary property is this: let $X = \sum_{i=1}^n X_i$. Then

$$E\left[\sum_i X_i\right] = \sum_i E[X_i]$$

Examples. Let S be a set with $|S| = n$. Also $1 \le r \le n$.

- Random r-sets: $\Omega = \binom{S}{r}$ and $\Pr(R) = \frac{1}{\binom{n}{r}}$ if $R \in \Omega$.
- Random permutations: $\Omega = S!$ and $\Pr(\pi) = \frac{1}{n!}$ if $\pi \in \Omega$.
- Random subsets: $\Omega = 2^S$. Let $0 < \alpha < 1$ and $\Pr(R) = \alpha^r(1-\alpha)^{n-r}$ for all $R \in \Omega$ with $|R| = r$. This corresponds to a random subset of S by picking any element with probility α. When $\alpha = 1/2$, then $\Pr(R) = 2^{-n}$ for all R.

Useful Facts.

- Harmonic numbers

$$\begin{aligned} H_n &:= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \\ &= \ln n + \Theta(1) \end{aligned}$$

- Let $0 \le x < 1$. Then

$$\begin{aligned} e^{-x} &= 1 - x + \frac{x^2}{2} - \frac{x^3}{3!} + \cdots \\ &\ge 1 - x \end{aligned}$$

- Consider the probability space above where $\Omega = 2^S$. If R is a random element of Ω, what is the expected size $|R|$? We can compute as follows:

$$\begin{aligned} E[|R|] &= \sum_{r=0}^{n} \sum_{R \in 2^S, |R|=r} r\alpha^r(1-\alpha)^{n-r} \\ &= \sum_{r=1}^{n} \binom{n}{r} r\alpha^r(1-\alpha)^{n-r} \\ &= n\alpha \sum_{r=1}^{n} \binom{n-1}{r-1} \alpha^{r-1}(1-\alpha)^{n-r} \\ &= n\alpha \sum_{r=0}^{n-1} \binom{n-1}{r} \alpha^{r}(1-\alpha)^{n-1-r} \\ &= n\alpha. \end{aligned}$$

4 Another View of Quicksort

The standard QuickSort algorithm can be described as follows: let S be a set of n points, and we want to return the sorted sequence $QSort(S)$.
0. If $|S| \le 1$ return S.
1. Pick a random $x_0 \in S$.

2. Partition S into $S_L \uplus \{x_0\} \uplus S_R$ where $S_L = \{y \in S : y < x_0\}$ and $S_R = \{y \in S : y > x_0\}$.
3. Return $(QSort(S_L), x_0, QSort(S_R))$.

In each of the above problems:

- We are given a set S of n objects (= points, segments, planes). We solve the problem on S_r where S_r is a random r-subset of S. Let
$$S_r = \{x_1, \ldots, x_r\}, \qquad T_r := S - S_r.$$
- View this as the problem of maintaining a data structure D_0 such that after the rth stage, D_0 is a representation a suitable mathematical object $D(S_r)$. The algorithm amounts to inserting updating D_0 so that it represents $D(S_{r+1})$. Thus, D_0 is a semi-dynamic datastructure.
- Besides D_0, we typically must to maintain an auxilliary data structure D_1 (conflict graphs, history graph, etc) which depends on the problem.
- There are two versions of semi-dynamic: whether what we maintain depends on the set $T_r = S - S_r$ or not. If so, then the problem is not online.

We illustrate these ideas in one dimension. The standard QuickSort algorithm can be described as follows: let S be a set of n points, and we want to return the sorted sequence $QSort(S)$.
0. If $|S| \leq 1$ return S.
1. Pick a random $x_0 \in S$.
2. Partition S into $S_L \uplus \{x_0\} \uplus S_R$ where $S_L = \{y \in S : y < x_0\}$ and $S_R = \{y \in S : y > x_0\}$.
3. Return $(QSort(S_L), x_0, QSort(S_R))$.

In our new take on the Quicksort algorithm, we assume an iterative algorithm where in the rth step ($r = 1, \ldots, n$), we want to insert an element $x_r \in S$ into a data structure. Let $S_r = \{x_1, \ldots, x_r\}$ and $D(S_r)$ be the data structure after we have inserted the first $r - 1$ elements. Assume $D(S_r)$ is a linear list of all the intervals between successive numbers in S_r. If
$$S_r : y_1 < y_2 < \cdots < y_r$$
then the intervals are $I_1 = (y_1, y_2)$, $I_2 = (y_2, y_3)$, etc. Let $\mathcal{R}_0(S_r)$ be the set of intervals, including $(-\infty, y_1]$ and $[y_r, +\infty)$. For simplicity, we assume all inputs numbers are distinct.

Data Structures. The main data structure is just a list D_0 representing the intervals of $\mathcal{R}_0(S_r)$. With suitable representation, we can easily obtain the sorting order of S_r from D_0.

The auxiliary data structure is a list D_1 comprising all the numbers in T_r. Each $y \in T_r$ points to the interval $I \in \mathcal{R}_0(S_r)$ that contains it. Furthermore, we augment D_0 so that with each $I \in \mathcal{R}_0(S_r)$, we store list $C(I)$ of all those numbers $y \in T_r$ such that $y \in I$. Call $C(I)$ the **conflict list** for I.

Update. When we insert a new x_{r+1}, we will split some interval $I \in \mathcal{R}_0(S_r)$ into two subintervals. Updating the list D_0 is easy: just replace I by two intervals. To update D_1, we must split the adjacency list for I in the obvious way.

Analysis. We analyze the expected cost of the rth update. We use this opportunity to introduce the backwards analysis technique of P. Chew and popularized by R. Seidel. Namely, imagine running the algorithm backwards. Instead of inserting x_r, we actually delete x_r from the currently sorted sublists, and merge the two intervals which shares x_r as a common endpoint to obtain the data structure corresponding to S_{r-1}. Let t_r be the cost of this deletion. Now, there are r points that can be deleted, and they are all equally likely. There are also $n-r$ numbers in T_r whose pointers may have to be adjusted. For $j = 1, \ldots, n-r$, let $I_j = I_{j,r}$ be the indicator variable such that $I_j = 1$ if the pointer of the jth number is adjusted by the rth deletion, and otherwise $I_j = 0$. Up to a constant factor we may write,

$$t_r = 1 + \sum_{j=1}^{n-r} I_j$$

Hence

$$\begin{aligned} E[t_r] &= 1 + \sum_{j=1}^{n-r} E[I_j] \\ &= 1 + \sum_{j=1}^{n-r} \Pr[I_j = 1] \\ &\le 1 + \sum_{j=1}^{n-r} 2/r \end{aligned}$$

since the pointer of any number in T_r is changed only if we delete the one or two numbers that bound its interval. Thus

$$E[t_r] \le 1 + 2(n-r)/r \le 1 + 2n/r.$$

The expected overall cost of the algorithm is therefore

$$\sum_{r=1}^{n} E[t_r] = n + \sum_{r=1}^{n} 2n/r = n + 2nH_n = O(n \log n).$$

where H_n is the harmonic number $\sum_{i=1}^{a} 1/i = \ln n + \Theta(1)$.

REMARK: why do we call this "quicksort"? The central idea of quicksort is that the partition element is chosen randomly from the input. The usual presentation of Quicksort uses a partition function to split the remaining elements into two sets, and this corresponds to our update of the conflict lists. While the usual view proceeds recursively, we have avoided this completely by a "flat" view of the data. This should have the practical benefit of avoiding the overhead of recursion.

An Online Version. The previous algorithm is not online because our data structure requires knowledge of the set T_r. Let us introduce another key idea in randomized incremental algorithms: the **history data structure**. This idea first appeared in a paper of Boissonnat and Teillaud for computing the Delaunay Triangulation of a point set.

Notice that the intervals of $\cup_{i=1}^{r}\mathcal{R}_0(S_i)$ are organized into a binary search tree D_2 in a natural way: the root is the interval $[-\infty, +\infty]$. In general, if interval I is split into I_L and I_R by an insertion, then we let I_L and I_R be the left and right child of I in the binary tree. Thus, the intervals of $\mathcal{R}_0(S_r)$ are represented by the intervals of the leaves of T_r. Suppose we are given x_r to be inserted into $\mathcal{R}_0(S_{r-1})$. We first use D_2 to find the interval $I_r \in \mathcal{R}_0(S_{r-1})$ that must be split. We then proceed as before.

What is the complexity of this algorithm? We must now bound the cost of searching in D_2. Let $t = t_r$ be the length of the search path. Here we must introduce an assumption that $(x_1, \ldots, x_r)$ is a random permutation of S_r and so t is a random variable. Let V_r be the expected value of t_r. Then,

$$V_r = 1 + \frac{1}{r+1}\sum_{i=0}^{r} V_i.$$

Multiplying by $r+1$ and differencing,

$$V_r = \frac{1}{r} + V_{r-1}$$

and hence $V_r = H_r$ (the rth Harmonic number.

5 Configuration Spaces

We are now going to abstract the properties of this algorithm using the concept of **configuration spaces**. The elements of configuration spaces are as follows:

- There is a set of **objects** $\mathcal{O}$. For sorting, $\mathcal{O} = \mathbb{R}$. Let $p \in \mathcal{O}$ be a typical object.
- The **input instance** for our problem is a finite set $S \subseteq \mathcal{O}$ of objects.
- There is a set of **regions** $\mathcal{R}$. For sorting, $\mathcal{R}$ is the set of intervals. Let $A \in \mathcal{R}$ be a typical region.
- There is a finite set $\Delta := \{\Delta_i : i = 1, \ldots, b\}$ of **defining functions**. Each Δ_i has the form
$$\Delta_i : \mathcal{O}^{d_i} \to \mathcal{R}$$
where d_i is the **degree** of Δ_i.

  If $\Delta_i(p_1, \ldots, p_{d_i}) = A$, then we say A is **defined** by $p_1, \ldots, p_{d_i}$. Furthermore, every region in $\mathcal{R}$ is defined by some sequence of objects. The maximum degree of any Δ_i is called the **degree** of Δ.

For sorting, $b = 3$. We have $\Delta_1(p, q) = (p, q)$ is the open interval with endpoints $p, q \in \mathbb{R}$. Also $\Delta_2(p) = (p, \infty)$ and $\Delta_3(p) = (-\infty, p)$. The degree of Δ is 2.

- There is a **conflict relation** $\mathcal{K} \subseteq \mathcal{O} \times \mathcal{R}$. If $\mathcal{K}(p, A)$ holds, we say that p conflicts with A.

  For sorting, $\mathcal{K}(p, A)$ holds if $p \in A$.

- Let $\Pi(S)$ be the set of regions defined by the finite set $S \subseteq \mathcal{O}$. For sorting,

$$\Pi(S) = \Delta_1(S^2) \cup \Delta_2(S) \cup \Delta_3(S).$$

  We also define the subset $\Pi_0(S)$ comprising those regions in $\Pi(S)$ that has no conflicts in S.

- We frame our computational problem on input S to be the computation of the set $\Pi_0(S)$.

  For sorting, we want to compute all those intervals formed by consecutive numbers in S.

Convex Hull. Let us apply this to the convex hull problem (P2) where the input is a set of points in $\mathbb{R}^d$.

- The set of object is $\mathcal{O} = \mathbb{R}^d$. Let $p \in \mathcal{O}$ be a typical object.
- THe set of regions $\mathcal{R}$ is the set of open half-spaces in $\mathbb{R}^d$.
- There is only one defining function, $\Delta := \{\Delta_1\}$ $(b = 1)$. Moreover, Δ_1 has degree d,

$$\Delta_1 : \mathcal{O}^d \to \mathcal{R}$$

  which defines a open half-space in $\mathbb{R}^d$. In case the points $p_1, \ldots, p_d$ are degenerate, $\Delta_1(p_1, \ldots, p_d) = \emptyset$ (the empty set).

- The **conflict relation** $\mathcal{K} \subseteq \mathcal{O} \times \mathcal{R}$ says that $\mathcal{K}(p, A)$ holds iff $p \in A$.
- The input instance for our problem is a finite set $S \subseteq \mathcal{O}$ of points. It defines a set $\Pi(S)$ of open half-spaces.
- The convex hull problem amounts to computing the half-spaces in $\Pi_0(S)$. (In case of degeneracy, we can resolve the ambiguities in any suitable way).

Trapezoidal Decomposition. We consider next the trapezoidal decompostion (P3):

- The set of object is $\mathcal{O} = (\mathbb{R}^2)^2$, corresponding to line segments.
- The set of regions $\mathcal{R}$ is the set of closed quadrilaterals with two verical sides. We allow degeneration into triangles.

- We have MANY defining function, $\Delta := \{\Delta_1, \ldots, \Delta_b\}$: $b = b_1 + b_2 + b_3 + b_4 = 4 + 2 + 2 + 1$.

  FIGURE

  where we show three cases only.

- The conflict relation $\mathcal{K}(p, A)$ holds iff the segment p intersects the quadrilateral A.

- The input instance for our problem is a finite set $S \subseteq \mathcal{O}$ of segments (note that we do not assume that they are disjoint).

- The trapezoidal decomposition problem amounts to computing the empty regions of $\Pi_0(S)$.

6 FINAL REMARKS

Other Applications of These ideas:
(a) Randomized Linear Programming: $O_d(n)$ time
(b) Output Sensitive Convex Hull: $O(n^2 + k \log n)$.

References:

(1) Mulmuley: Computational Geometry: An Introduction Through Randomized Algorithms, Prentice-Hall, 1994
(2) Boissonnat and Yvinec: Algorithmic Geometry, Cambridge University Press (1998), (English Translation: H.Bronnimann).

Exercises

Exercise 0.1: Solve the following recurrence exactly: for $a, b \in \mathbb{N}$ let

$$\begin{aligned} W_{a,b} &= 1 + \frac{1}{a+b}\left(\sum_{i=0}^{a-1} W_{i,b} + \sum_{j=0}^{b-1} W_{a,j}\right), \\ (a+b)W_{a,b} &= (a+b) + \sum_{i=0}^{a-1} W_{i,b} + \sum_{j=0}^{b-1} W_{a,j}. \end{aligned}$$

Assume the boundary condition $W_{0,0} = 1$. ♢

End Exercises