Randomized Algorithms (INFO-F413)

Jean Cardinal

2024

Contents

Foreword

This document is intended to be used as lecture notes for the course INFO-F413 on Randomized Algorithms.

The goal of this course is to convey the theoretical foundations of the applications of randomness and probabilities in the design of efficient algorithms, and a provide a hands-on experience on the programming of randomized algorithms.

It is a striking and mysterious phenomenon that randomness, and the capability for a computing device to make random moves, seems to be a useful resource in computation. We will illustrate this on a number of elementary applications ranging from sorting and data structures to graph and optimization algorithms.

Two major references will be used throughout the course:

MR *Randomized Algorithms*, Rajeev Motwani and Prabhakar Raghavan (Cambridge U. Press, 1995)

MU *Probability and Computing*, Michael Mitzenmacher and Eli Upfal (Cambridge U. Press, 2012)

Chapter 1

Probabilities

1.1 Axioms

A *probability space* is made of three components:

1. A sample space Ω of the set of possible outcomes,
2. a family $\mathcal{F}$ of sets representing the events,
3. a probability function $P : \mathcal{F} \to [0, 1]$.

The probabillity function must satisfy the two following axioms:

1. $P(\Omega) = 1$
2. for any (possibly infinite) sequence of pairwise disjoint events $E_1, E_2, \ldots$:

$$P\left(\bigcup_{i\geq 1} E_i\right) = \sum_{i\geq 1} P(E_i).$$

1.2 Integer random variables

In what follows, the probability spaces we consider will usually be *discrete*, hence Ω is either finite or countably infinite. We will consider *discrete random variables*, defined as functions from a set of outcomes Ω to $\mathbb{N}$. The probability of a random variable X having some value x is

$$P(X = x) = \sum_{s\in\Omega:X(s)=x} P(s).$$

For example, let X be the random variable giving the sum of two independent dice throws. We have

$$P(X = 4) = P((1, 3)) + P((2, 2)) + P((3, 1)) = 3/36 = 1/12.$$

Two random variables X and Y are said to be *independent* whenever the following holds:

$$P(X = x \wedge Y = y) = P(X = x) \cdot P(Y = y).$$

The *expectation* of a discrete random variable is

$$E[X] := \sum_{i \in \mathbb{N}} i \cdot P(X = i).$$

We will often use the following equality for a collection of discrete random variables $\{X_i\}$, which is known as the *linearity of expectation*.

Proposition 1

Let $\{X_i\}$ be a collection of discrete random variables. Then

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

In particular, it implies that $E[cX] = cE[X]$.

For two (possibly not independent) random variables, we can define the *conditional probability* $P(X = x|Y = y)$. The definition of the *conditional expectation* follows:

$$E[X|Y = y] = \sum_x x \cdot P(X = x|Y = y).$$

For instance, consider the sum of the values of two dice $X = X_1 + X_2$. Then $E[X|X_1 = 2] = \sum_x x \cdot P(X = x|X_1 = 2) = \sum_{x=3}^{8} x\frac{1}{6} = \frac{33}{6} = \frac{11}{2}$. Also note that the expression $E[X|Y]$ denotes a random variable, whose value is $E[X|Y = y]$ whenever $Y = y$.

The *variance* σ_X^2 of a discrete, bounded random variable X is the expected squared deviation from its expectation:

$$\sigma_X^2 = E[(X - E[X])^2]$$

1.3 Union bound

Although completely elementary, the following inequality is a precious tool in many situations.

Proposition 2

Let E_1 and E_2 be two events in Ω. Then

$$P(E_1 \cup E_2) \leq P(E_1) + P(E_2)$$

1.4 Famous discrete distributions

The *Bernoulli* distribution is perhaps the most elementary discrete distribution, which can be interpreted as a biased coin flip. It is defined by

$$P(X = 1) = p,$$

and

$$P(X = 0) = 1 - p,$$

for some parameter $p \in [0, 1]$. The variance of a Bernoulli random variable is $p(1 - p)$.

The *binomial* distribution is defined as:

$$P(X = j) = \binom{n}{j} p^j (1 - p)^{n-j}.$$

This can be interpreted as giving the probability of having exactly j heads and $n - j$ tails in n consecutive throws of a biased coin, where p is the probability of a heads. In other words, it is the sum of n independent Bernoulli variables of parameter p. Let us denote by X_i the outcome of the ith variable. We have

$$\begin{aligned} X &= \sum_{i=1}^{n} X_i \\ E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= np. \end{aligned}$$

The *geometric* distribution is defined by

$$P(X = j) = (1 - p)^{j-1} p.$$

It can be interpreted as giving the number of throws of a biased coin before and including the

first heads. We can compute its expectation as follows:

$$\begin{aligned} E[X] &= \sum_{j=1}^{\infty} j \cdot (1-p)^{j-1} p \\ &= p \sum_{j=1}^{\infty} j \cdot (1-p)^{j-1} \\ &= p \frac{d}{dp} \left[-\sum_{j=0}^{\infty} (1-p)^j \right] \\ &= p \frac{d}{dp} \left[-\frac{1}{p} \right] \\ &= \frac{1}{p} \end{aligned}$$

1.5 Useful bounds

Proposition 3

The following holds for all integers $n > 0$:

$$\left(1 - \frac{1}{n}\right)^n < \frac{1}{e}. \tag{1.1}$$

It is obtained starting from a simple observation:

$$\begin{aligned} 1 - x &\leq e^{-x} \\ 1 - \frac{1}{n} &\leq e^{-\frac{1}{n}} \\ \left(1 - \frac{1}{n}\right)^n &< \frac{1}{e}. \end{aligned}$$

This bound is tight in the following (well-known) sense:

$$\lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}, \tag{1.2}$$

Harmonic numbers are sums of the inverses of the first n integers, and appear in countless analyses.

Proposition 4

Define

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

We have

$$H_n \simeq \ln n + \gamma + o(1),$$

where $\gamma \simeq 0.5772156649$ is the Euler–Mascheroni constant.

Exercise 1

A ship arrives at a port, and the n sailors on board go ashore for revelry. Later at night, the n sailors return to the ship and, in their state of inebriation, each independently chooses a random cabin to sleep in. Show that the probability that no sailor returns to her own cabin approaches $1/e$ as n grows large.

Solution of Exercise 1

The probability that a sailor does not return to her own cabin is $1 - 1/n$. Since the choices are independent, the probability that none of the n sailors return to her own cabin is $(1 - 1/n)^n$. The limit is $1/e$ as proved above.

Exercise 2

Suppose you are given a coin for which the probability of heads, say p, is unknown. How can you use this coin to generate *unbiased* coin flips? Give a scheme for which the expected number of flips of the biased coin for extracting one unbiased flip is $1/(p(1-p))$. **Hint:** consider two consecutive flips.

Solution of Exercise 2

First observe that if you perform two successive flips of the coin, the probability that you get heads followed by tails is the same as that of having tails followed by heads. This probability is equal to $p(1-p)$. The scheme uses this symmetry, and is the following. Perform two successive flips. If you get heads followed by tails, output heads. If you get tails followed by heads, output tails. Otherwise (that is, if you get twice heads or twice tails), repeat.

The number of iterations is the expectation of a geometric random variable of parameter $2p(1-p)$, the probability of getting distinct outcomes from the two throws. The expectation is $1/(2p(1-p))$. Since we flip two coins at every iterations, the number of flips is as claimed.

Exercise 3

Consider a Monte-Carlo algorithm A running in time polynomial in the input size n, and whose probability of *success* is $1/n^k$ for some positive integer k. Prove that we can use it to design another polynomial-time algorithm whose probability of *error* is *exponentially* small (say e^{-n}). What is its running time?

Solution of Exercise 3

Let us denote by $T(n)$ the running time of A. We consider t independent executions of A. The probability that A fails for all t executions is at most

$$\left(1 - 1/n^k\right)^t.$$

Let us set t to n^{k+1}. The probability of failure is now

$$\begin{aligned}\left(1 - 1/n^k\right)^t &= \left(1 - 1/n^k\right)^{n^{k+1}} \\ &= \left(\left(1 - 1/n^k\right)^{n^k}\right)^n \\ &\leq e^{-n}.\end{aligned}$$

The running time is now $T(n) \cdot n^{k+1}$, which remains polynomial.

Exercise 4

The birthday Paradox. Consider m balls that are thrown independently and uniformly at random in one of n distinct bins. What is the probability that no two balls land in the same bin? Explain the birthday paradox.

Solution of Exercise 4

Let us throw the balls one at a time and let E_i be the event that the ith ball lands in an empty bin. We can observe that $P(E_1) = 1$ and $P(E_2) = 1 - 1/n$. The probability that no two balls land in the same bin is

$$\begin{aligned}P(E_1)P(E_2|E_1)P(E_3|E_1 \cap E_2)\ldots P(E_i|\bigcap_{j<i} E_j)\ldots P(E_m|\bigcap_{j<n} E_j) &= \prod_{i=1}^{m}\left(1 - \frac{i-1}{n}\right) \\ &\leq \prod_{i=1}^{m} e^{-(i-1)/n} \\ &= e^{-m(m-1)/(2n)}.\end{aligned}$$

Hence with $m \simeq \sqrt{2n} + 1$ this happens with probability at most $1/e$. The birthday paradox is for $n = 365$: then with $m > 28$, two people have their birthdays on the same date with probability at least $1 - 1/e \simeq 0.632$. For 70 people, the probability is greater than 0.998.

Exercise 5

Let $X_1, X_2, \ldots, X_m$ be independent random variables. Let $X = \sum_{i=1}^{m} X_i$. Prove that $\sigma_X^2 = \sum_{i=1}^{m} \sigma_{X_i}^2$.

Solution of Exercise 5

Let $E[X] = \mu$ and $E[X_i] = \mu_i$. We have

$$\sigma_X^2 = E[(X-\mu)^2] = E[(\sum_{i=1}^{m}(X_i - \mu_i))^2] \quad (1.3)$$

$$= \sum_{i=1}^{m} E[(X_i - \mu_i)^2] + 2 \cdot \sum_{i<j} E[(X_i - \mu_i)(X_j - \mu_j)]. \quad (1.4)$$

Because the X_i are independent, we can split the second term as follows:

$$= \sum_{i=1}^{m} E[(X_i - \mu_i)^2] + 2 \cdot \sum_{i<j} E[X_i - \mu_i]E[X_j - \mu_j]. \quad (1.5)$$

Now we can observe that $E[X_i - \mu_i] = E[X_i] - \mu_i = 0$ for all i. Hence only the first term remains:

$$\sigma_X^2 = \sum_{i=1}^{m} E[(X_i - \mu_i)^2] = \sum_{i} \sigma_{X_i}^2. \quad (1.6)$$

Chapter 2

Quicksort and selection

Quicksort is a canonical example of a divide-and-conquer algorithm, and the basis of countless investigations in analytic combinatorics and computer science. A slick analysis of its running time is given here, together with an extension to randomized selection.

See textbook: Mitzenmacher & Upfal, Chapter 2.5.

2.1 Quicksort

We can obtain a very nice and quick analysis of the Quicksort algorithm using indicator variables and Harmonic numbers. We consider the simple randomized version of QuickSort for sorting n pairwise comparable objects, where the pivot is chosen uniformly at random.

Theorem 1

The expected number of comparisons performed by Quicksort is $2n \ln n + O(n)$.

Suppose we sort n elements denoted by $x_1, x_2, \ldots, x_n$, so that x_i has rank i in the sorted order. We define the random variable X_{ij} for $1 \leq i < j \leq n$ as the number of times that x_i is compared to x_j during the execution of the algorithm. Note that no such pair can be compared more than once, therefore $P(X_{ij} = 1) = E[X_{ij}]$. The total number of comparisons is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

It is not too difficult to realize that $P(X_{ij} = 1)$ is exactly equal to the probability that **the first pivot chosen in the interval** $x_i, x_{i+1}, \ldots, x_j$ **is either** x_i **or** x_j. Since the number of

elements in the interval is $j - i + 1$, this probability is exactly equal to $\frac{2}{j-i+1}$. Therefore,

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{k=2}^{n}(n-k+1)\frac{2}{k} \\ &= 2n(H_n - 1) - 2(n-1) + 2(H_n - 1) \\ &= 2n\ln n + O(n). \end{aligned}$$

2.2 Selection

We can use the same strategy for analyzing the randomized selection algorithm, the analogue of QuickSort for finding the element of rank k in the underlying linear order. There we recurse only on the part containing the rank k element, unless the sought element is the pivot. We assume without loss of generality that $k \leq n/2$. The analysis is due to Eppstein (2007).

Theorem 2

The expected number of comparisons performed by the randomized selection algorithm is

$$2n + 2k\ln\left(\frac{n-k}{k}\right) + 2n\ln\left(\frac{n}{n-k}\right)$$

.

We can define the same indicator variable X_{ij} as for QuickSort, but the expression of its expectation is a bit more complicated:

$$E[X_{ij}] = \frac{2}{\max\{k-i+1, j-k+1, j-i+1\}}.$$

Indeed:

1. In the case where $k > j$, if the first pivot chosen in the interval $\{x_i, \ldots, x_j, \ldots, x_k\}$ is neither x_i nor x_j, then the elements x_i and x_j will never be compared.
2. In the case where $k < i$, if the first pivot chosen in the interval $\{x_k, \ldots, x_i, \ldots, x_j\}$ is neither x_i nor x_j, then the elements x_i and x_j will never be compared.

3. Finally, in the case where $i < k < j$, if the first pivot chosen in the interval $\{x_i, \ldots, x_k, \ldots, x_j\}$ is neither x_i nor x_j, then the elements x_i and x_j will never be compared.

We are now left with summing this quantity for all pairs i, j such that $1 \leq i < j \leq n$. Figure 2.1 explains how we can split the sum into five parts. The diagonals on the picture have identical values $2/q$ for $E[X_{ij}]$, where

$$q = \max\{k - i + 1, j - k + 1, j - i + 1\}.$$

More precisely:

1. For the bottom triangle, the vertical segments have the same value $2/q$ with $q = k - i + 1$. The qth segment has q terms in it, hence the sum is $\sum_{q=1}^{k} q \cdot 2/q = 2k$.

2. For the rightmost triangle, the sum is the same, only with k replaced by $n - k$ (and q is now $j - k + 1$). Hence this is $2(n - k)$.

3. When $q = j - i + 1$, we have three subcases. In all cases, pairs along the diagonals have identical values.

   (a) The bottom triangle corresponds to the case where $q \leq k$. There are again k diagonals, and the qth diagonal has approximately q entries, each of value $2/q$. Hence the sum is again approximately $2k$.

   (b) The parallelogram is the region where $k \leq q \leq n - k$. There we have k terms in each diagonal, each of the form $2/q$:

$$\sum_{q=k}^{n-k} k\frac{2}{q} = 2k \sum_{q=k}^{n-k} \frac{1}{q} = 2k(H_{n-k} - H_k).$$

   (c) Finally, the topmost triangle corresponds to the values of q such that $q \geq n - k$. The qth diagonal has $(n - q)$ terms, each of value $2/q$. We have the sum:

$$\sum_{q=n-k}^{n} (n - q)\frac{2}{q} = \sum_{q=n-k}^{n} \frac{2n}{q} + \sum_{q=n-k}^{n} (-2) = 2n(H_n - H_{n-k}) - 2k.$$

We use $H_{n-k} - H_k \sim \ln \frac{n-k}{k}$, and $H_n - H_{n-k} \sim \ln \frac{n}{n-k}$. Overall, the whole sum evaluates to

$$2n + 2k \ln\left(\frac{n-k}{k}\right) + 2n \ln\left(\frac{n}{n-k}\right).$$

For $k = n/2$, this simplifies to

$$\sim 2n(\ln 2 + 1).$$

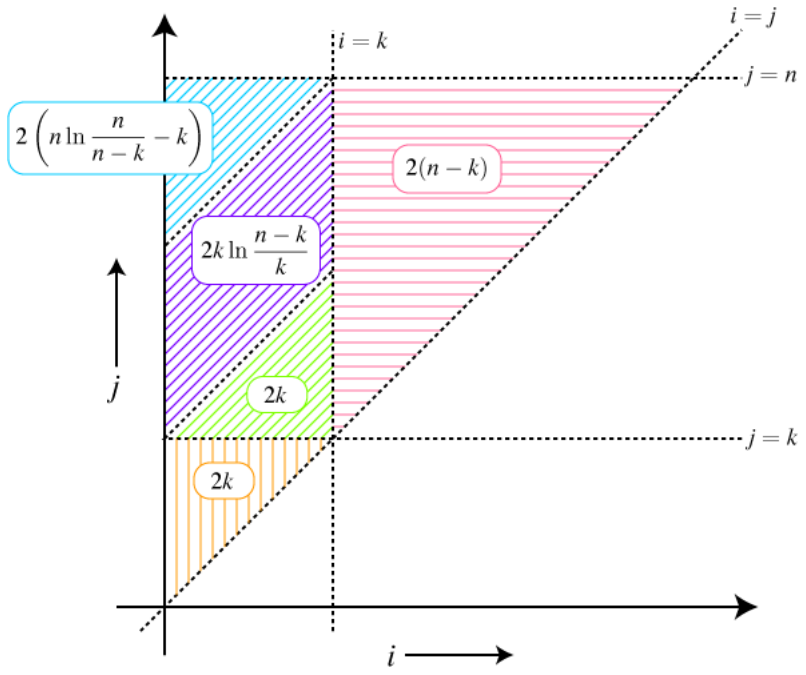

Figure 2.1: How to compute the sum in the indicator-based analysis of the randomized selection algorithm (Eppstein 2007).

The expression can be simplified by setting $k = 1 + \alpha(n-1)$ for some $\alpha \in [0,1]$. The *entropy* of a Bernoulli random variable of parameter α is

$$h(\alpha) = (1-\alpha) \ln \frac{1}{1-\alpha} + \alpha \ln \frac{1}{\alpha}.$$

The expected number of comparisons of the randomized selection algorithm can be written as:

$$2n + 2k \ln \left(\frac{n-k}{k}\right) + 2n \ln \left(\frac{n}{n-k}\right) \sim 2n(h(\alpha)+1).$$

Since $h(\alpha)$ takes values in $[0, \ln 2]$, we have that the worst case is indeed reached for $k = n/2$.

Chapter 3

Fingerprinting

Fingerprinting refers to a family of simple algebraic techniques for testing equality between elements of a large universe. Deciding whether two elements x and y drawn from a universe U can be done exactly in time $\log |U|$, assuming that we can use a total order on U. The universe U can be very large, though. The idea of fingerprinting is to map the two elements x and y randomly to a smaller set V, so that if their images in V are equal, the probability of x and y being equal is high enough.

See textbook: Motwani & Raghavan, Chapter 7.

3.1 Polynomial identity testing

Suppose we are given two degree-d polynomials $F(x)$ and $G(x)$ in two different forms: $F(x)$ is given as a product $\prod_{i=1}^{d}(x-a_i)$, and $G(x)$ is given in its canonical form (a list of monomials with their coefficients). We would like to decide quickly whether the polynomials are identical, that is, whether $F(x) \equiv G(x)$. Transforming F into its canonical form requires $\Theta(d^2)$ operations.

We can use randomization to perform this checking in an efficient way. The algorithm is as follows:

1. pick a number r at random in the range $\{1, 2, \ldots, 100d\}$,
2. If $F(r) = G(r)$ return Yes, otherwise return No.

Computing the values of $F(r)$ and $G(r)$ can be done in time $O(d)$. If $F(x) \not\equiv G(x)$, it may still be the case that the algorithm gives the wrong answer. This happens when r is a root of $F(x) - G(x)$. But the degree of this new polynomial is at most d, hence by the fundamental theorem of algebra, there can be at most d roots. Since there are $100d$ possible values for r, there is only a probability at most $d/(100d) = 1/100$ that this happens. By

repeating the algorithm k times with independent random picks, the probability of error is at most $(1/100)^k$.

Proposition 5

There is a randomized Monte-Carlo algorithm for testing equality between two degree-d polynomials by evaluating them k times, that succeeds with probability at least $1 - (1/100)^k$.

Note that if we proceed without replacement, that is, if we avoid using the same number r twice, we obtain a probability of failure that is slightly better:

$$\prod_{j=1}^{k} \frac{d-(j-1)}{100d-(j-1)}.$$

Note that d iterations are necessary to have a zero probability of error, but then the algorithm is not more efficient than using the standard approach.

The generalization to n-variate polynomials involves the Schwartz-Zippel Theorem.

Theorem 3

Let $Q(x_1, x_2, \ldots, x_n)$ be a multivariate polynomial of degree d in the field $\mathbb{F}$. Fix any finite set $S \subset \mathbb{F}$ and let $r_1, r_2, \ldots, r_n$ be picked at random from S. Then

$$P(Q(r_1, r_2, \ldots, r_n) = 0 | Q(x_1, x_2, \ldots, x_n) \not\equiv 0) \leq \frac{d}{|S|}.$$

Polynomial identity testing, in which the access to a polynomial is only via evaluation, is a famous problem in computer science, not known to be solvable in deterministic polynomial time.

3.2 Equality of strings

Consider now the problem of deciding whether two binary strings $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ are equal. We can of course check each bit individually, which would cost n comparisons. Instead, we can try to map the two strings to a smaller set of values. Let us first interpret the two strings as numbers, by letting

$$a = \sum_{i=1}^{n} 2^{i-1} a_i, \text{ and } b = \sum_{i=1}^{n} 2^{i-1} b_i.$$

Let

$$F_p(x) = x \mod p$$

for a small *prime number* p.

The fingerprinting fails when $F_p(a) = F_p(b)$, but $a \neq b$. This happens only when p divides $c = |a - b|$, and $c \neq 0$. The idea is to pick a *random* prime p in the range $[0, tn \log(tn)]$, for some large t. We then use the following simple observation.

Proposition 6

The number of prime divisors of numbers less than 2^n is at most n.

Furthermore, the Prime Number Theorem tells us that

Theorem 4

The number $\pi(k)$ of distinct primes less than k is asymptotically $k/\ln k$.

Together, we have

Proposition 7

$$P(F_p(a) = F_p(b) | a \neq b) \leq \frac{n}{\pi(tn \log(tn))} = O\left(\frac{1}{t}\right).$$

3.3 Rabin-Karp string matching algorithm

The *string matching problem* is the following: Given two binary strings $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_m$, with $m < n$, decide whether Y appears in X as a substring. There is a trivial algorithm that simply checks every possible position of a substring, which takes time $O(nm)$.

We can apply the fingerprinting technique above, and instead look for a substring

$$X(j) = x_j x_{j+1} \dots x_{j+m-1}$$

such that $F_p(X(j)) = F_p(Y)$, where p is a random prime.

Suppose that p is chosen to be at most τ. Then the probability of a false positive at position j is

$$P(F_p(X(j)) = F_p(Y) | X(j) \neq Y) \leq \frac{m}{\pi(\tau)} = O\left(\frac{m \log \tau}{\tau}\right).$$

By choosing $\tau = n^2 m \log(n^2 m)$, we obtain that the probability that a false match occurs (at any position) is at most $O(1/n)$.

It remains to evaluate the complexity of the algorithm. The trick here is to realize that the fingerprint need not be recomputed at each step. The following relation holds:

$$F_p(X(j+1)) = 2(F_p(X(j)) - 2^{m-1} x_j) + x_{j+m} \mod p.$$

Applying this observation, we can check that the whole algorithm can be made to run in time $O(n + m)$.

Proposition 8

The Rabin-Karp string matching algorithm runs in $O(n + m)$ time and has a probability of error of $O(1/n)$.

Exercise 6

From Monte-Carlo to Las Vegas Give a Las Vegas algorithm for the string matching problem running in expected time $O(n + m)$.

Chapter 4

Hashing

Hashing is perhaps the simplest way to organize a *dictionary* data structure. We have a collection of pairs composed of a key and some attached information, and we wish to be able to quickly find the information attached to a given key, and update the collection by removing or adding pairs. When the keys are totally ordered, binary trees and the like allow to recover the information in time logarithmic in the number of elements. This is best possible in the information-theoretic sense, hence in the comparison-based model.

If we are allowed to perform arithmetic operations on the keys, then we can obtain dictionaries with constant expected access time. To analyze hashing schemes properly, randomization is in order.

> **See textbook:** Motwani & Raghavan, Chapter 8.4, and Mitzenmacher & Upfal, Chapter 13.3.

4.1 Universal hashing

Suppose we wish to build a hash table based on a hash function $h : M \to N$, where $M = \{0, 1, \ldots, m-1\}$ and $N = \{0, 1, \ldots, n-1\}$, with $m > n$. In order to randomize a hash table, we are going to pick a *random hash function*.

A family H of hash functions is said to be *2-universal* if for all $x \neq y \in M$, and for h a function chosen uniformly at random in H, we have

$$P(h(x) = h(y)) \leq \frac{1}{n}.$$

Note that this probability if the same as for a random function. Hence picking a random function in a 2-universal family yields the same pairwise collision probability as a random function. Note that the hash function is chosen once and for all at the creation of the table, and used subsequently for all operations, until the table is deleted or rebuilt.

Define the following indicator functions (remember capital letters denote sets):

$$\delta(x, y, h) = \begin{cases} 1 \text{ if } \ h(x) = h(y) \text{ and } x \neq y \\ 0 \text{ otherwise.} \end{cases}$$

$$\begin{aligned}
\delta(x, y, H) &= \sum_{h \in H} \delta(x, y, h) && (4.1)\\
\delta(x, Y, h) &= \sum_{y \in Y} \delta(x, y, h) && (4.2)\\
\delta(X, Y, h) &= \sum_{x \in X} \delta(x, Y, h) && (4.3)\\
\delta(x, Y, H) &= \sum_{y \in Y} \delta(x, y, H) && (4.4)\\
\delta(X, Y, H) &= \sum_{x \in X} \delta(x, Y, H). && (4.5)
\end{aligned}$$

If we pick our hash function from a universal family of hash functions, we can bound the expected size of a bucket in a hash table with *chaining*, in which every location is associated with a list containing all the keys that were mapped to this location.

Proposition 9

Let H be a 2-universal family. For all $x \in M, S \subseteq M$, and random hash function $h \in H$, we have

$$E[\delta(x, S, h)] \leq \frac{|S|}{n}.$$

The proof is direct from calculation:

$$\begin{aligned}
E[\delta(x, S, h)] &= \sum_{h \in H} \frac{\delta(x, S, h)}{|H|} \\
&= \frac{1}{|H|} \sum_{h \in H} \sum_{y \in S} \delta(x, y, h) \\
&= \frac{1}{|H|} \sum_{y \in S} \delta(x, y, H) \\
&= \frac{1}{|H|} \sum_{y \in S} \frac{|H|}{n} \\
&= \frac{|S|}{n}.
\end{aligned}$$

We directly get the following running time for a hash table with chaining.

Theorem 5

For any sequence of r operations (insert, delete, search) in a hash table with chaining, with s inserts and h chosen at random from a 2-universal family, the total expected cost is at most

$$r\left(1+\frac{s}{n}\right).$$

Hence if the size n of the table is chosen to be larger than the maximum number s of elements at any time in the table, the expected total cost per operation is at most 2!

4.2 Universal families

Exercise 7

A universal family Let p be a large prime greater than m. We have $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ and we define

$$\begin{aligned} g(x) &= x \mod n \\ f_{a,b}(x) &= ax + b \mod p \\ h_{a,b}(x) &= g(f_{a,b}(x)). \end{aligned}$$

Consider $H = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0\}$.

1. Prove that $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.
2. Prove that H is 2-universal.

Solution of Exercise 7

1. Let $f_{a,b}(x) = r$ and $f_{a,b}(y) = s$. Observe that $r \neq s$, since $a \neq 0$ and $x \neq y$. The parameters a and b that lead to these values are uniquely defined by the following system of equations:

$$ax + b = r \mod p \tag{4.6}$$
$$ay + b = s \mod p \tag{4.7}$$

which has a unique solution over the field $\mathbb{Z}_p$. There is a collision if and only if $g(r) = g(s)$, hence when $r = s \mod n$. Hence the number of hash functions $h_{a,b}$ in H that cause x and y to collide is exactly the number of choices of $r \neq s$ such that $r = s \mod n$, hence $\delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

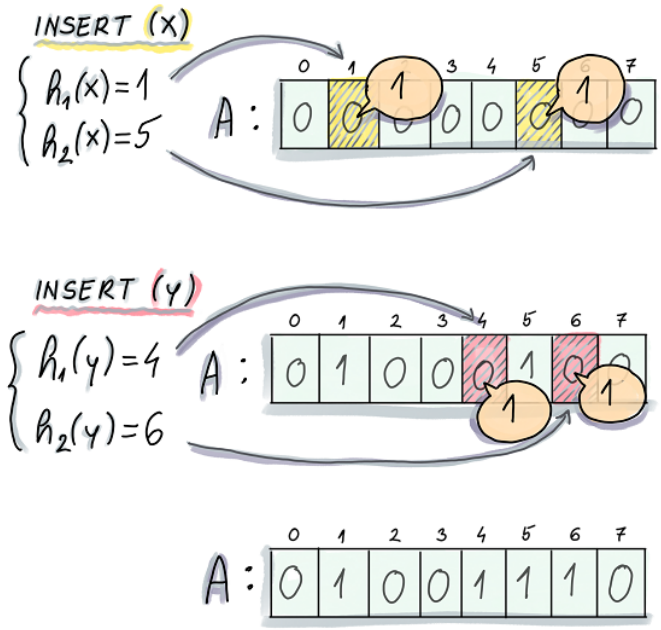

Figure 4.1: Illustration of a Bloom filter update for $k = 2$. From *Algorithms and Data Structures for Massive Datasets* by Dzejla Medjedovic, Emin Tahirovic, and Ines Dedovic, Manning 2022.

2. For every $r \in \mathbb{Z}_p$, there are at most $\lceil p/n \rceil$ different choices of $s \in \mathbb{Z}_p$ such that $r = s \bmod n$. Hence
$$\delta(\mathbb{Z}_p, \mathbb{Z}_p, g) \leq p\,(\lceil p/n \rceil - 1)) \leq \frac{p(p-1)}{n}.$$
Now since $|H| = p(p-1)$, we have from the previous point that $\delta(x, y, H) \leq |H|/n$, and H is 2-universal.

4.3 Bloom filters

See textbook: Mitzenmacher & Upfal, Chapter 5.5.3.

When we only need to remember whether an element is *present or not* in a set, and do not care for any other information, we can simply set a bit to one in an array of bits. If this array is as large as the universe M of keys, we maintain this information perfectly. Otherwise, we can use a hash function to determine the index of the bit to flip. This leads to a *false positive* problem, where an element seems to be present, while the bit was flipped by another element with the same hash value.

In order to reduce the rate of false positives, we can use k hash functions, denoted by $h_1, h_2, \ldots, h_k$, with range $1, 2, \ldots, n$, where n is the size of our array of bits. What is the value of k that minimizes the false positive probability? Let us denote by s the number of elements in our set.

Proposition 10

The value of k that minimizes the error probability for a Bloom filter of size n and a set of s elements is approximately

$$\lceil (-\ln 2)\frac{n}{s} \rceil.$$

Proof. For the sake of analysis, we make the simplifying assumption that the hash values are random and uniformly distributed. When an element $x \in M$ is inserted in the set, we set all the bits of indices $h_1(x), h_2(x), \ldots, h_k(x)$ to one.

When inserting the s elements, we have set ks bits to 1. Hence the probability that a specific bit remains 0 is

$$\left(1 - \frac{1}{n}\right)^{ks} \sim e^{-ks/n},$$

(where we use $\left(1 - \frac{1}{n}\right)^n \sim e^{-1}$, see first lecture). We set $p := e^{-ks/n}$. The probability of a false positive is now

$$f := (1-p)^k.$$

In order to minimize the probability of a false positive, we can derive this expression with respect to k and find the minimum. There is, however, a way to avoid those calculations. In order to see this, we define k as a function of p, and optimize with respect to p instead. We have

$$k = (-\ln p)\frac{n}{s}.$$

Hence

$$\begin{aligned} f &= (1-p)^k \\ &= (1-p)^{(-\ln p)\frac{n}{s}} \\ &= (e^{-\ln(p)\ln(1-p)})^{n/s}. \end{aligned}$$

This expression is minimized whenever $p(1-p)$ is maximized, which happens exactly for $p = 1/2$. Therefore, the best value of k is

$$k^* := (-\ln 2)\frac{n}{s}.$$

Of course, k must be integer, so we may need to pick the floor or ceiling of this expression. □

Chapter 5

Complexity classes

P is the class of decision problems for which there exists an algorithm running in worst-case polynomial time. NP is the class of decision problems for which there exists a *nondeterministic* algorithm running in worst-case polynomial time. The word algorithm in those definitions refer to a *deterministic* algorithm, defined, for instance, as a Turing machine. Suppose we augment a Turing machine with the ability to flip coins. Does this allow to solve some problems more efficiently? This complexity-theoretic question is far from being closed. We will take a quick tour of a couple of definitions of complexity classes involving randomized algorithms.

See textbook: Motwani & Raghavan, Chapter 1.5.

5.1 RP and coRP

The class RP (Randomized Polynomial-time) consists of languages L such that there exists a *randomized* algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts with probability $\geq 1/2$.
- $x \notin L \Rightarrow A(x)$ rejects.

It is not difficult to realize that $\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$.

Similarly, the class coRP consists of languages L such that there exists a *randomized* algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts.
- $x \notin L \Rightarrow A(x)$ rejects with probability $\geq 1/2$.

The class ZPP (Zero-error Probabilistic Polynomial-time) consists of languages having a *Las Vegas* algorithm running in *expected* polynomial time.

In the following two exercises, we will prove the following alternative characterization of ZPP:

Theorem 6

$\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$.

Exercise 8

Prove that $\mathsf{RP} \cap \mathsf{coRP} \subseteq \mathsf{ZPP}$.

Solution of Exercise 8

Suppose that a language L is in $\mathsf{RP} \cap \mathsf{coRP}$. Then there exist two algorithms A and B such that:

- if $x \in L$, then $A(x)$ accepts with probability $\geq 1/2$, and $B(x)$ always accepts,
- if $x \notin L$, then $A(x)$ always rejects, and $B(x)$ rejects with probability $\geq 1/2$.

Now we can run both algorithms on an input x. If $A(x)$ accepts, then we know for sure that $x \in L$, and we accept. Similarly, if $B(x)$ rejects, we know for sure that $x \notin L$, and we reject. Otherwise, when $A(x)$ rejects and $B(x)$ accepts, we iterate. It cannot be the case that $A(x)$ always rejects and $B(x)$ always accepts. Hence after an expected constant number of iterations, we will know the answer for sure.

Exercise 9

Prove that $\mathsf{ZPP} \subseteq \mathsf{RP} \cap \mathsf{coRP}$.

Solution of Exercise 9

Suppose that a language belongs to ZPP. We now exhibit two algorithms, one showing that the language also lies in RP, and one showing it lies in coRP.

For the first case, suppose that the ZPP algorithm A runs in expected polynomial time $p(n)$, where $n = |x|$. Then we do the following:

- run the algorithm A for $2p(n)$ steps,
- if A has stopped after $2p(n)$ steps, then output its answer, otherwise, reject.

What is the probability that A stops after $2p(n)$ steps, knowing that it stops *on expectation* after $p(n)$ steps? From Markov's inequality, letting X be the running time of A, we have $Pr[X > 2p(n)] \leq 1/2$. Hence if $x \in L$, with probability at least $1/2$, we have the correct answer. If $x \notin L$, then we always reject. This proves $L \in \mathsf{RP}$.

Similarly, one can prove that $L \in \mathsf{coRP}$ by accepting the input whenever A does not stop after $2p(n)$ steps.

5.2 Two-sided error

Let us make a first attempt at defining the class of problems that have a Monte-Carlo algorithm with *two-sided error*.

The class PP (Probabilistic Polynomial-time) consists of languages L such that there exists a *randomized* algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts with probability $> 1/2$.
- $x \notin L \Rightarrow A(x)$ accepts with probability $< 1/2$.

A problem with this definition is that we cannot amplify the success probability if the error probability is too close to $1/2$.

A more practical definition of two-sided error Monte-Carlo algorithms is the following. The class BPP (Bounded-error Probabilistic Polynomial-time) consists of languages L such that there exists a *randomized* algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$,

- $x \in L \Rightarrow A(x)$ accepts with probability $\geq 3/4$.
- $x \notin L \Rightarrow A(x)$ accepts with probability $\leq 1/4$.

Exercise 10

Prove that the definition of BPP does not change if instead of the error probability $1/4$, we take any number $p < 1/2$.

Solution of Exercise 10

Let the error probabilities be $1-p$ and p, respectively instead of $3/4$ and $1/4$, with $p < 1/2$. Let us show that this probability can be amplified by repeating the algorithm many times, and outputting the majority answer. Note that unlike in the case of one-sided error, the majority rule is our best bet. When repeating the algorith $2m+1$ times, the probability of error is the probability of having at most m correct answers. This error probability can thus be bounded as follows:

$$\begin{aligned}
\sum_{i=0}^{m} \binom{2m+1}{i} (1-p)^i p^{2m+1-i} &\leq \sum_{i=0}^{m} \binom{2m+1}{i} (1-p)^m p^{m+1} \\
&= (1-p)^m p^{m+1} \sum_{i=0}^{m} \binom{2m+1}{i} \\
&= (1-p)^m p^{m+1} 2^{2m} \\
&\leq (4p(1-p))^m,
\end{aligned}$$

where in the first line we used that $p < 1 - p$.

Note that in the above we only used that $p < 1 - p$, hence that $p = 1/2 - \varepsilon$ for some positive ε. This is sometimes given as the definition of BPP. Also note that the expression $(4p(1-p))^m$ of the obtained error probability can be made arbitrary close to, say, $1/2^n$, where n is the size of the input. Indeed, we have

$$(4p(1-p))^m = 1/2^n \Rightarrow m = O(n).$$

So we have the following statement.

Proposition 11

The class BPP consists of languages such that there exists a *randomized* algorithm A running in worst-case polynomial time and with a two-sided error probability of at most $1/2^n$ on an input of size n.

A famous open problem is to decide whether P = BPP, hence whether randomization helps in solving more problems in polynomial time. It is not even known whether BPP $\subseteq$ NP. Yet, a positive inclusion result about BPP will be given in the form of Adleman's Theorem.

Chapter 6

Derandomization

There is no known systematic way to get rid of randomness in an algorithm while keeping it efficient. There is a way to make a randomized algorithm deterministic, but at the price of introducing a novel feature, namely the access to a polynomial-sized "advice" that depends only on the size n of the input. This feature is also known as *nonuniformity*.

See textbook: Motwani & Raghavan, Chapter 2.3.

6.1 Polynomial advice and Adleman's theorem

Let $a(n)$ be a function of the integers to strings in Σ^*. We say that a deterministic algorithm A decides a language L with advice a if on an input $x \in \Sigma^*$, it uses the read-only string $a(|x|)$, where $|x|$ is the input size, to decide whether $x \in L$. The advice function a is not computed, it depends only on the size of the input, and is given to the algorithm for free. This type of algorithm is known as "nonuniform", because it can behave wildly differently depending on the size of the input. "Uniform" algorithms, on the other hand, are the ones we are familiar with, and run similarly for inputs of different sizes.

We define the class P/poly of languages that can be decided in polynomial time by a deterministic machine with an advice function a such that $a(n)$ is bounded by a polynomial in n. The following statement is one version of Adleman's Theorem.

Theorem 7

BPP $\subseteq$ P/poly.

Proof. We consider a two-sided error polynomial-time Monte-Carlo algorithm A for a language L, and show that there exists an advice function that can be used by a deterministic algorithm to decide L in polynomial time. We suppose without loss of generality that the input alphabet is $\Sigma = \{0, 1\}$, so that there are 2^n possible inputs of size n.

First observe that the error probability can be made exponentially vanishing with respect to the input size n, as shown in the previous lecture. Let us assume without loss of generality that the error probability of A is at most $1/2^{n+1}$ (any function smaller than $1/2^n$ by a constant factor will do).

Suppose now that algorithm A uses r random bits (that is, it flips a coin r times) to decide on an input of size n. Note that A runs in worst-case polynomial time, hence r is bounded by a polynomial in n.

Let us consider the $2^n \times 2^r$ Boolean matrix indicating, for each pair composed of a string of n input bits and a string of r random bits, whether A gives the correct answer for this input, having used these random bits. Since the error probability is at most $1/2^{n+1}$, the total number of pairs for which the answer is incorrect is at most

$$2^n \times 2^r/2^{n+1} = 2^r/2.$$

Since this is strictly less than 2^r, it implies that there exists a column of the matrix, hence a string $a(n)$ of r bits, such that using those bits instead of random ones, the answer is correct *on all 2^n possible inputs*! Hence an algorithm that simulates the behavior of A but using the string $a(n)$ as advice can also decide L in polynomial time. □

6.2 Polynomial advice and circuits

A *Boolean circuit* with n inputs is a directed acyclic graph with the following properties:

1. There are n *input vertices* of indegree 0, and one *output vertex* of outdegree 1.
2. Every vertex that is not an input vertex is labeled by a Boolean operation in $\{\vee, \wedge, \neg\}$, and $\neg$ vertices have indegree 1.
3. Every input vertex is assigned a Boolean value, and other vertices compute the corresponding Boolean operation on their incoming edges and assign the result to their outgoing edges.
4. The *size* of a circuit is its number of vertices.

The circuit *computes* a Boolean function of the input values, and the result is the value returned by the output vertex.

A sequence $C_1, C_2, \ldots$ of circuits is said to be a *polynomial-sized circuit family* for a language L if for every n, the circuit C_n computes $x \in L$ for all $x \in \Sigma^n$. The set of languages having a circuit family of size $O(n^c)$ is denoted $\mathsf{SIZE}(n^c)$, and the set of languages having a polynomial-sized circuit family is therefore $\bigcup_{c>0} \mathsf{SIZE}(n^c)$.

We now observe that a language is in $\mathsf{P/poly}$ if and only if it has a polynomial-sized family of circuits.

Proposition 12

$\mathsf{P/poly} = \bigcup_{c>0} \mathsf{SIZE}(n^c)$

Proof. One direction is easy: if there exists a polynomial-sized family, then the advice $a(n)$ can consist in a description of the circuit C_n, which is then run by an algorithm for simulating a Boolean circuit.

On the other hand, first observe that if there exists a deterministic algorithm that decides L in polynomial time (hence if $L \in \mathsf{P}$), then there exists a polynomial-sized family of circuits for L. Polynomially many steps of the execution of a Turing machine, for instance, can be encoded as a circuit.

Now if there exists an algorithm that in addition uses a polynomial-sized advice string $a(n)$, we can construct a polynomial-sized family of circuits where the advice $a(n)$ is wired directly in the circuit. And since $a(n)$ has size polynomial in n, the sizes of these circuits are also polynomial in n. □

Chapter 7

Game trees and Yao's principle

From basic notions on zero-sum game theory, one can infer a way to prove lower bounds on the complexity of randomized algorithms. This is illustrated on a nice toy problem, consisting of evaluating the output of a simple tree-shaped circuit.

See textbook: Motwani & Raghavan, Chapters 2.1 and 2.2.

7.1 Game tree evaluation

The $\min - \max$ trees are the basis of many artificial intelligence algorithms for two-player games such as chess. Here we study the simplest incarnation of those, in which all values are binary and the min and max nodes correspond respectively to the Boolean $\wedge$ ("and") and $\vee$ ("or").

We consider a rooted complete binary tree T_k of even height $2k$, and whose internal nodes are labeled either $\wedge$ or $\vee$ depending on whether they are on an odd or even level. The root is labeled $\wedge$. Thus any root to leaf path starts with an $\wedge$ node and goes through exactly k $\wedge$ nodes and k $\vee$ nodes. Every internal node performs the Boolean operation corresponding to its label on its two children and pass the result to its parent. The $2^{2k} = 4^k$ leaves are labeled by Boolean values (for convenience, 0 or 1), playing the role of the input. The root is the output node. If we orient every edge of the tree towards the root, we obtain a special kind of Boolean circuit.

The *evaluation problem* is to decide the value of the root, given the values of the leaves. The complexity measure we will use is the number of leaves that are read by the algorithm. We first observe that in the deterministic case, any algorithm can be forced to read all 4^k leaves.

Exercise 11

Show that for any deterministic algorithm, there is an instance of the game tree evaluation problem that forces the algorithm to read the values of all of the 4^k leaves.

Solution of Exercise 11

(Recall that in the statement of the problem, we assumed the root node was an $\wedge$ node.)

We proceed by induction on k. For the purpose, we prove a stronger statement, that there are two instances that force the algorithm to read all leaves, with respective values 0 and 1. The base case is with $k = 0$, in which the value is a constant given as input.

Suppose that it holds for $k - 1$. We prove that it holds for k as well.

First we describe the instance of value 1. In that case, the values of both $\vee$ children of the root must be 1. One of the $\wedge$ children of one such $\vee$ node is always evaluated first by the deterministic algorithm. By the induction hypothesis, there are two instances of values 0 and 1 and of height $2(k-1)$ requiring the algorithm to read all leaves. We can assign the instance of value 0 to the first evaluated subtree, and the instance of value 1 to the other, forcing the deterministic algorithm to evaluate both subtrees, and therefore all leaves.

For the instance of value 0, we can suppose that the $\vee$ subtree that is evaluated first by the deterministic algorithm has value 1, and do the same as above. This forces the algorithm to evaluate the other $\vee$ subtree, which must have value 0. This in turn forces the algorithm to evaluate both $\wedge$ subtrees with value 0. By induction, there exist such subtrees that force the algorithm to read all leaves again.

With randomization, however, we can improve on this lower bound. A simple randomized algorithm works as follows. Recursively evaluate one of the two children of the current node (starting from the root) chosen at random. If the current node is an $\wedge$ node, and the outcome of the recursive call is 0, there is no need to check the other child, and we output 0. Similarly, if the current node is an $\vee$ node and the output of the recursive call is 1, we do not need to check the other child and we can also safely return 1. In all the other cases, we need to perform a second recursive call on the other child to output the correct value.

The fact that the first child on which we recurse is chosen at random is enough to defeat the adversary designed in the above exercise. We can show the following.

Proposition 13

The expected number of leaves read by the simple randomized recursive strategy for game tree evaluation is at most 3^k.

Proof. We proceed by induction on k. The base case is for $k = 1$.

Now for the induction step, suppose that the result holds for $k - 1$, hence that the randomized algorithms read in expectation no more than 3^{k-1} leaves of T_{k-1}.

Suppose that the current node is an $\vee$ node that returns 0. Then certainly we need two recursive calls, which cost, from the induction hypothesis, $2 \times 3^{k-1}$ expected reads. If the $\vee$ node returns 1, then with probability at least $1/2$, we only need to check one child. Hence the expected cost is

$$\frac{1}{2} \times 3^{k-1} + \frac{1}{2} \times 2 \times 3^{k-1} = \frac{3}{2} \times 3^{k-1}.$$

Suppose now that the current node is an $\wedge$ node. If it evaluates to 1, then both its children must return 1, and the expected cost of the two recursive calls is

$$2 \times \frac{3}{2} \times 3^{k-1} = 3^k.$$

If it evaluates to 0, then it can be either the case that both $\vee$ children return 0, and the expected cost is

$$2 \times 3^{k-1} < 3^k,$$

or that the $\vee$ children return respectively 0 and 1, in which case the expected cost is

$$\frac{1}{2} \times 2 \times 3^{k-1} + \frac{1}{2} \times \left(\frac{3}{2} \times 3^{k-1} + 2 \times 3^{k-1} \right) = \frac{11}{4} \times 3^{k-1} < 3^k.$$

Hence in all cases, the expected number of reads to evaluate the root $\wedge$ node is no more than 3^k. □

The input length can be taken to be the number of leaves $n = 4^k$. We therefore have $k = \log_4 n$ and $3^k = 3^{\log_4 n} = n^{\log_4 3} \simeq n^{0.793}$. Hence this is a sublinear-time algorithm!

7.2 Game theory

We now introduce some actual game theory, as pioneered by Von Neumann and Morgenstern ini the 1930s.

A *zero-sum two-player* game can be represented by an $n \times m$ payoff matrix M, whose entries are indexed by a pair of strategies, one for each player, and M_{ij} is the amount payed by the column player C to the row player R when R chooses strategy i and C chooses strategy j.

Exercise 12

Show that the following inequality is valid for all payoff matrices:

$$\max_i \min_j M_{ij} \leq \min_j \max_i M_{ij}$$

Solution of Exercise 12

Consider an arbitrary element M_{ij}. Clearly, we have

$$\min_k M_{ik} \leq M_{ij} \leq \max_\ell M_{\ell,j}.$$

Hence

$$\forall i, j : \ \min_k M_{ik} \leq \max_\ell M_{\ell,j}.$$

In particular

$$\max_i \min_k M_{ik} \leq \min_j \max_\ell M_{\ell,j},$$

which is equivalent to the original statement.

We now consider *mixed strategies*, defined as probability distributions on the rows (for the row player) or columns (for the column player) of the matrix M. The expected payoff for a pair p, q of strategies (respectively, on the rows and columns) is

$$E[\text{payoff}] = p^T M q = \sum_{i=1}^{n} \sum_{j=1}^{m} p_i M_{ij} q_j.$$

Let us denote by V_R the best lower bound on the expected payoff to the column player by choosing a mixed strategy p, and by V_C the best upper bound on the payoff paid by the column player for a mixed strategy q. Hence

$$V_R = \max_p \min_q p^T M q \tag{7.1}$$

$$V_C = \min_q \max_p p^T M q. \tag{7.2}$$

The Minmax Theorem of Von Neumann states that those two values are always equal.

Theorem 8

$$\max_p \min_q p^T M q = \min_q \max_p p^T M q.$$

7.3 Yao's Minmax principle

Given a computational problem Π, consider a finite collection $\mathcal{A}$ of algorithms and a finite set $\mathcal{I}$ of inputs, such that all algorithms in $\mathcal{A}$ correctly solve the problem Π on all inputs in $\mathcal{I}$. Denote by $C(I, A)$ the running time of algorithm $A \in \mathcal{A}$ on input $I \in \mathcal{I}$.

This defines a zero-sum two-player game between a algorithm designer and an adversarial input. The set of algorithms can be seen as a set of strategies for the algorithm designer, and a mixed strategy as a randomized algorithm. Similarly, a mixed strategy for the input player is a distribution on the possible inputs. We can apply Von Neumann's Theorem to this game:

Proposition 14

For probability distributions p and q on $\mathcal{I}$ and $\mathcal{A}$, respectively, let I_p be corresponding random input and A_q the corresponding randomizd algorithm. Then

$$\max_p \min_q E[C(I_p, A_q)] = \min_q \max_p E[C(I_p, A_q)].$$

and

$$\max_p \min_{A \in \mathcal{A}} E[C(I_p, A)] = \min_q \max_{I \in \mathcal{I}} E[C(I, A_q)].$$

The second line is obtained by observing that the second-level optimization can be restricted to pure strategies. Yao deduced the following statement, known a Yao's Minmax principle.

Theorem 9

For all distributions p over $\mathcal{I}$ and q over $\mathcal{A}$, we have

$$\min_{A \in \mathcal{A}} E[C(I_p, A)] \leq \max_{I \in \mathcal{I}} E[C(I, A_q)].$$

In other words, the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution p is a lower bound on the expected running time of any Las Vegas randomized algorithm for Π.

7.4 Lower bound for game tree evaluation

We can apply Yao's principle to the game tree evaluation problem. For any chosen distribution on the input, the expected running time of the best deterministic algorithm will be a lower bound on the expected running time of any randomized algorithm.

We first simplify the definition of the problem.

Exercise 13

Show that the tree T_k, of height $2k$, is equivalent to a tree T'_k in which all internal nodes are labeled by the $\neg\vee$ ("nor") function (hence return 0 unless both inputs are 0).

Now let each leaf be set to 1 independently with probability

$$p = \frac{3 - \sqrt{5}}{2}.$$

This value may seem to come out of nowhere, but has been chosen for a specific purpose, namely the following property.

Exercise 14

Prove that every internal $\neg\vee$ node of T'_k returns 1 with the same probability p.

We omit the proof of the technical statement that we can restrict our attention to deterministic algorithm that proceed by *depth-first pruning*, that is, by recursively evaluating subtrees, starting from the root.

Among those, we can realize that no specific algorithm can be better than another, since the distribution on the input and the shape of the tree are completely symmetric. Therefore, the expected number of leaves $W(h)$ read by any such deterministic algorithm on a tree of height h can be written as:

$$W(h) = W(h-1) + (1-p)W(h-1),$$

since we have to recurse a second time only if the first call returned 0, which happens with probability $1-p$. The recurrence can be solved directly as follows (taking $W(0) = 1$):

$$\begin{aligned} W(h) &= W(h-1) + (1-p)W(h-1) \\ &= (2-p)W(h-1) \\ &= (2-p)^h \\ &= (2-p)^{\log_2 n} \\ &= n^{\log_2(2-p)} \\ &\simeq n^{0.694}. \end{aligned}$$

From Yao's principle, we therefore deduce the following:

Proposition 15

The expected number of leaves read by any exact randomized algorithm for game tree evaluation is $\Omega(n^{0.694})$.

Note that this does not allow us to conclude whether our simple randomized $O(n^{0.793})$ strategy is optimal. What is your bet?

Chapter 8

Concentration bounds

Chernoff bounds allow to make statements about how far a sum of random variables is likely to be from its expected values. This will reveal useful in situations where we want to prove that an algorithm behaves well not only in expectation, but with high probability. We prove the Chernoff bounds from first principles, starting from the most elementary inequality.

See textbook: Motwani & Raghavan, Chapters 3.2 and 4, Mitzenmacher & Upfal, Chapter 4.

8.1 Markov's inequality

Theorem 10

If X is a nonnegative random variable, and $a > 0$, then

$$P(X \geq a) \leq \frac{E[X]}{a}.$$

Note that, equivalently, for any $k > 1$,

$$P(X \geq k \cdot E[X]) \leq \frac{1}{k}.$$

Proof. Define a function

$$f(x) = \begin{cases} 1 & \text{if } x \geq a \\ 0 & \text{otherwise.} \end{cases}$$

Then $P(X \geq a) = E[f(X)]$. Since $f(x) \leq x/a$ for all x, we have

$$E[f(X)] \leq E\left[\frac{X}{a}\right] = \frac{E[X]}{a}.$$

□

8.2 Chebyshev's inequality

Theorem 11

Let X be a random variable with expectation μ_X and standard deviation σ_X. Then for any $t \in \mathbb{R}^+$,

$$P(|X - \mu_X| \geq t\sigma_X) \leq 1/t^2.$$

Proof. Observe that

$$P(|X - \mu_X| \geq t\sigma_X) = P((X - \mu_X)^2 \geq t^2\sigma_X^2).$$

Now the random variable $Y = (X - \mu_X)^2$ has expectation $E[Y] = \sigma_X^2$. We can thus apply Markov's inequality above with $k = t^2$. □

8.3 Chernoff bounds

Theorem 12

Let $\{X_i\}_{i=1}^n$ be a collection of independent Bernoulli random variables, where $X_i \in \{0, 1\}$ and $P(X_i = 1) = p_i$ for some $0 < p_i < 1$. Let $X = \sum_i X_i$, and set $\mu = E[X] = \sum_i p_i$. Then for all positive δ,

$$P(X > (1 + \delta)\mu) < F(\delta, \mu),$$

where

$$F(\delta, \mu) := \left(\frac{e^\delta}{(1 + \delta)^{(1+\delta)}}\right)^\mu.$$

Proof. For any positive real t, we have

$$P(X > (1 + \delta)\mu) = P(e^{tX} > e^{t(1+\delta)\mu}).$$

Now from Markov's inequality applied to right-hand side, we obtain:

$$P(X > (1 + \delta)\mu) < \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}}.$$

(We can assume a strict inequality here, since the X_i are not identically 0 or 1.) Now observe that

$$E[e^{tX}] = E[e^{t\sum_i X_i}] = E\left[\prod_i e^{tX_i}\right] = \prod_i E[e^{tX_i}],$$

where the last equality is from the independence assumption on the X_i. We now have

$$P(X > (1 + \delta)\mu) < \frac{\prod_i E[e^{tX_i}]}{e^{t(1+\delta)\mu}}.$$

Now $E[e^{tX_i}] = p_i e^t + 1 - p_i = 1 + p_i(e^t - 1)$ by definition. Hence

$$\begin{aligned}\prod_i E[e^{tX_i}] &= \prod_i (1 + p_i(e^t - 1)) \\ &< \prod_i e^{p_i(e^t-1)} \\ &= e^{\sum_i p_i(e^t-1)} \\ &= e^{(e^t-1)\mu},\end{aligned}$$

where we used the inequality $1+x < e^x$ at the second line. Plugging this back in our inequality yields

$$P(X > (1+\delta)\mu) < \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}}.$$

We are still free to choose a suitable value of t that gives us the best bound. We can check that this happens for $t = \ln(1+\delta)$, which gives us the claimed bound. □

*
* *

Exercise 15

Simplified Chernoff bounds.

1. Prove the logarithmic inequality $\ln(1+x) \geq \frac{x}{1+x/2}$ for all $x > 0$.
2. Apply this result to the (logarithm of the) right-hand side of Chernoff's inequality and deduce that
$$F(\delta, \mu) \leq e^{-\frac{\delta^2}{2+\delta}\mu}.$$

Solution of Exercise 15

1. We can study the function $f(x) = \ln(1+x) - \frac{x}{1+x/2}$. Its derivative is
$$f'(x) = \frac{1}{1+x} - \frac{1}{(1+x/2)^2},$$
which cancels at $f'(x) = 0 \leftrightarrow (1+x/2)^2 - (1+x) = 0 \leftrightarrow x = 0$. We can then check that $f(0) = \ln 1 = 0$, hence that $f(x) \geq 0$ for all $x > 0$.
2. We have
$$\begin{aligned}\ln\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu &= \mu(\delta - (1+\delta)\ln(1+\delta)) \\ &\leq \mu\left(\delta - (1+\delta)\cdot\frac{\delta}{1+\delta/2}\right) \\ &= -\frac{\delta^2}{2+\delta}\mu.\end{aligned}$$

As a corollary, we obtain the following simpler formulation of Chernoff's upper bound.

Proposition 16

If $\delta > 2$, then

$$F(\delta, \mu) \leq e^{-\frac{\delta}{2}\mu}.$$

If $\delta < 2$, then

$$F(\delta, \mu) \leq e^{-\frac{\delta^2}{4}\mu}.$$

Note that so far we have bounded from above the probability that the sum of the Bernoulli variables were a factor $(1+\delta)$ *larger* than its expected value. A somehow symmetric statement exists to bound from above the probability that the sum is a factor $(1-\delta)$ *smaller* (for some $\delta \leq 1$) than its expected value. We skip the proof of this result, but the technique is similar (apply Markov's inequality to an exponential function of the studied variable).

Theorem 13

Let $\{X_i\}_{i=1}^n$ be a collection of independent Bernoulli random variables, where $X_i \in \{0, 1\}$ and $P(X_i = 1) = p_i$ for some $0 < p_i < 1$. Let $X = \sum_i X_i$, and set $\mu = E[X] = \sum_i p_i$. Then for all $0 < \delta \leq 1$,

$$P(X < (1-\delta)\mu) < e^{-\frac{\delta^2}{2}\mu}.$$

8.4 Balls and bins

We consider a classical balls and bins problem, that one can easily relate to hashing questions. Suppose we throw n balls into n bins, uniformly and independently at random. Let us find a value m such that the probability of having more than m balls in the first bin (or any other fixed bin) is at most $1/n^2$.

Let us denote by Y the number of balls in the first bin. We can apply the Chernoff bound to the indicator variables X_i, where $X_i = 1$ when the ith ball lands in the first bin, which happens with probability $p_i = 1/n$. We clearly have $Y = \sum_i X_i$, and the expected value $\mu = E[Y] = \sum_i p_i = 1$. From the simplified expression $F(\delta, \mu) \leq e^{-\frac{\delta}{2}\mu}$, we can solve

$$e^{-\frac{\delta}{2}} \leq 1/n^2 \Rightarrow \delta \geq 4 \ln n,$$

which gives us $m \geq 1 + 4 \ln n$.

Proposition 17

With probability at least $1 - 1/n^2$, there are no more than $1 + 4 \ln n$ balls in the first bin.

Note that the approximation on the upper bound on F has an impact on our result for this problem. If instead, we solve the inequality with respect to the original bound $\frac{e^\delta}{(1+\delta)^{(1+\delta)}}$, we can get a better constant in front of the $\ln n$ term.

8.5 Set balancing

The following is a classical problem, often given as an application of the *probabilistic method* in combinatorics. Consider a set of n individuals, and m *binary characteristics*, such that each individual either has or does not have the characteristic (for instance, being older than some age, taller than some height, etc.) We wish to partition the individuals into two groups such that for each characteristic, there are roughly the same number of people having this characteristic in each group.

Consider an $m \times n$ binary matrix A, such that $A_{ij} = 1$ if and only if individual j has the characteristic i. The partition into two groups will be given by a vector $b \in \{-1, +1\}^n$, so that $b_j = +1$ if and only if the individual j is in the first group. We consider the *imbalance vector* $c = Ab \in \mathbb{Z}^n$. The ith component c_i is the difference between the number of people having characteristic i in the first group and in the second group. If $c_i > 0$, then there are c_i more people in the first group with characteristic i than in the second group, and vice-versa for $c_i < 0$.

What if we partition randomly? Let us choose a random vector b, with $P(b_j = +1) = P(b_j = -1) = 1/2$, independently, and *regardless of the matrix A*. Note that the expected value of c_i is exactly 0 for all i. In fact, with high probability, the maximum imbalance is bounded by $O(\sqrt{n})$.

Proposition 18

With probability at least $1 - O(1/n)$,

$$\max_i |c_i| \leq 4\sqrt{n \ln n}.$$

Proof. Consider the ith row A_i of A. The imbalance $c_i = A_i b$ is a sum of independent random variables in $\{0, -1, +1\}$.

We can safely ignore the 0 entries in A_i and consider a sum of $k_i \leq n$ variables in $\{-1, +1\}$, where k_i is the number of 1 entries in the row A_i. Let X_j be the jth such variable, for $1 \leq j \leq k_i$. We will need to make a change of variables so that the new variables are in $\{0, 1\}$ and we can apply the above bounds. Therefore, let us define $Y_j = (X_j + 1)/2$. We let $\mu = E[\sum_{j=1}^{k_i} Y_j] = k_i/2$. We have

$$
\begin{aligned}
P\left(c_i \geq 4\sqrt{n \ln n}\right) &= P\left(\sum_{j=1}^{k_i} X_j \geq 4\sqrt{n \ln n}\right) \\
&= P\left(\sum_{j=1}^{k_i} (2Y_j - 1) \geq 4\sqrt{n \ln n}\right) \\
&= P\left(\sum_{j=1}^{k_i} Y_j \geq 2\sqrt{n \ln n} + \mu\right) \\
&= P\left(\sum_{j=1}^{k_i} Y_j \geq \left(1 + \frac{2\sqrt{n \ln n}}{\mu}\right)\mu\right).
\end{aligned}
$$

We can then apply the Chernoff bound with $\delta = \frac{2\sqrt{n \ln n}}{\mu}$. Note that if $k_i < 2\sqrt{n \ln n}$, then the event we are considering has probability 0. Hence we can safely assume that $\delta < 2$, and obtain an upper bound of

$$
P\left(c_i \geq 4\sqrt{n \ln n}\right) < e^{-\frac{\delta^2}{4}\mu} = e^{-\frac{4n \ln n}{4\mu^2}\mu} = e^{-\frac{2n \ln n}{k_i}} \leq e^{-2 \ln n} = 1/n^2.
$$

With a similar development, we obtain

$$
P\left(c_i \leq -4\sqrt{n \ln n}\right) = P\left(\sum_{j=1}^{k_i} Y_j \leq \left(1 - \frac{2\sqrt{n \ln n}}{\mu}\right)\mu\right).
$$

We can then apply the Chernoff bound for the other direction and check the upper bound on

$$
P\left(c_i \leq -4\sqrt{n \ln n}\right) < e^{-\frac{\delta^2}{2}\mu} = e^{-\frac{4n \ln n}{2\mu^2}\mu} = e^{-\frac{4n \ln n}{k_i}} \leq e^{-4 \ln n} = 1/n^4.
$$

From the union bound, the probability that *any* of the imbalance c_i for $i = 1, 2, \ldots, n$ is larger than $4\sqrt{n \ln n}$ is at most $O(1/n)$. □

*
* *

The following exercises[1] present other typical applications of Chernoff bounds.

Exercise 16

We are given an array of n elements, and we are told that some element x occurs $2n/3$ times in the array, but we are not told the value of x. Our goal is to use a fast Monte-Carlo algorithm (that may report the incorrect value) to find x.

[1] Adapted from Pat Morin's, see https://cglab.ca/~morin/

1. Describe a constant-time Monte-Carlo algorithm to find x that is correct with probability $2/3$.

2. Suppose we sample k elements at random (with replacement) from the array. Give a good upper bound on the probability that x occurs less than $(1-\delta)2k/3$ times in this sample.

3. Give a good upper bound on the probability that x occurs less than $k/2$ times in the sample.

4. Describe a Monte-Carlo algorithm that runs in $O(k)$ time and reports x with probability at least $1 - 1/e^{\Omega(k)}$.

Exercise 17

Let $L = \ell_0, \ldots, \ell_{n+1}$ be a list of items. We find an "independent set" in L using the following algorithm: For each element $\ell_i, 1 \leq i \leq n$, we toss a fair coin. We say that ℓ_i is included in the independent set if ℓ_i's coin toss came up heads and both its neighbours' (ℓ_{i-1} and ℓ_{i+1}) coin tosses came up tails.

1. What is the probability that ℓ_i is included in the independent set? Using this, compute the expected size of the independent set.

2. Let E_i be the event "ℓ_i is included in the independent set." Are E_i and E_{i+1} independent?

3. Are E_i and E_{i+2} independent?

4. Are E_i and E_{i+3} independent?

5. Use Chernoff's bounds to show that, with very high probability, this algorithm produces an independent set of size at least cn for some constant $c > 0$.

Exercise 18

Imagine the following random walk on the integer grid. You begin by standing on the Euclidean plane at the origin $(0, 0)$ and then repeat the following n times: Toss a 4-sided die (see Figure 8.1) and depending on the result you either take a step north ($y \leftarrow y + 1$), east ($x \leftarrow x + 1$), south ($y \leftarrow y - 1$), or west ($x \leftarrow x - 1$).

1. At the end of this process, what is your expected x-coordinate and your expected y-coordinate?

2. Argue that, with high probability for any positive $\delta < 2$, the number of steps you take in any particular direction (to the west, say) is in the interval $[(1-\delta)n/4, (1+\delta)n/4]$.

Figure 8.1: 4-sided dice.

3. Argue that, with high probability, your random walk finishes inside a square of side length δn centered at the origin.

4. What is the smallest value of δ for which the above argument gives a meaningful result?

Solution of Exercise 18

1. Consider the x-axis (the reasoning for the y-axis is the same). The expected value of the final x coordinate is $n \times (1/4 \cdot (-1) + 1/4 \cdot 1) = 0$. Hence the expected (x, y) value is $(0, 0)$.

2. The number X of steps to the west is a sum of n independent Bernoulli random variables of parameter $1/4$. The expected value of X is $\mu = n/4$. The Chernoff bounds give us:

$$\begin{aligned} P[X < (1-\delta)n/4] &< e^{-(n/4)\delta^2/2} \\ P[X > (1+\delta)n/4] &< e^{-(n/4)\delta^2/4} \end{aligned}$$

Hence with probability at least $1 - 2e^{-n\delta^2/16}$ the number of steps to the west is in the indicated interval.

3. The above reasoning holds for all four directions. In particular, it holds for the number of steps to the east. Hence from the union bound, the probability that the random walk finishes with an x-coordinate in an interval of length δn centered at the origin is at least $1 - 4e^{-n\delta^2/16}$.

Similarly, the probability to end up in the square of side δn is at least $1 - 8e^{-n\delta^2/16}$.

4. For the result to make sense, we require that

$$\lim_{n\to\infty} 8e^{-n\delta^2} = 0$$

When $\delta \sim c/\sqrt{n}$ for some constant c, for instance, this does not hold. So to have actual high probabilities, we may require that δ is greater than some constant.

Chapter 9

Karger's minimum cut algorithm

As another simple application of randomized algorithms, we can consider the following graph-theoretic problem: Given a graph $G = (V, E)$ with n vertices, find a partition of the vertices into two parts such that the number of edges across the two parts is minimum. Such a set of edges is called a *cutset*, and the problem is known as the *min-cut* problem. We will use the word min-cut instead of "minimum-size cutset".

See textbook: Motwani & Raghavan, Chapter 10.2.

9.1 Baby version

We will analyze the following very simple procedure:

Algorithm Contract:

Input: A graph G.

Output: A cut C.

1. for i in $\{1, 2, \ldots, n-2\}$:
   (a) pick an edge e at random
   (b) contract e
2. output the set of edges connecting the two remaining vertices

A *contraction* of an edge e connecting vertices u and v merges the two vertices u, v into a single one, eliminates all edges connecting u and v, and retains all other edges in the graph. Note that we allow parallel edges (multiple edges between the same endpoints), but we cannot have self-loops (edges connecting a vertex to itself). In what follows, we will

Figure 9.1: Several runs of Karger's **Contract** algorithm.

assume that contracting a random edge takes time $O(n)$, so that the running time of the **Contract** algorithm is $O(n^2)$.

Before analyzing the success probability of **Contract**, we make two observations.

Proposition 19

In an n-vertex multigraph with min-cut value k, no vertex has degree smaller than k. Furthermore, the number of edges is at least $m \geq nk/2$.

Proposition 20

The min-cut value of a multigraph cannot decrease after the contraction of an edge.

Theorem 14

The **Contract** algorithm computes a min-cut with probability at least $2/(n(n-1))$.

Proof. Let C be a min-cut of size k, let E_i be the event that the edge contracted at iteration i is not in C, and let $F_i := \cap_{j=1}^{i} E_j$ be the event that no edge in the previous contraction steps is part of C.

From the first proposition above, the graph must have at least $nk/2$ edges. Therefore, the probability of *not* picking an edge of C satisfies $P(E_1) = P(F_1) \geq 1 - \frac{k}{\frac{nk}{2}} = 1 - \frac{2}{n}$.

In general, we have that

$$P(E_i|F_{i-1}) \geq 1 - \frac{k}{k(n-i+1)/2} = 1 - \frac{2}{n-i+1}.$$

The value we are interested in is $P(F_{n-2})$, which we can bound as follows:

$$\begin{aligned} P(F_{n-2}) &\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \\ &= \left(\frac{n-2}{n}\right) \cdot \left(\frac{n-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right) \\ &= \frac{2}{n(n-1)}. \end{aligned}$$

□

Again, we can reduce this probability by repeating the algorithm. If we repeat it $n(n-1)\ln n$ times, we obtain a probability of error of:

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\ln n} \leq e^{-2\ln n} = \frac{1}{n^2}.$$

In what follows, we will need to understand what happens when the contraction algorithm stops early.

Proposition 21

Supppose the **Contract** algorithm is terminated when the number of remaining vertices is exactly t. Then any specific min-cut survives in the contracted graph with probability at least

$$\binom{t}{2} / \binom{n}{2} = \Omega\left(\left(\frac{t}{n}\right)^2\right).$$

9.2 Improved

We will now try to improve on the running time. What if we simply terminate the **Contract** algorithm early, and use a slower but exact algorithm on multigraphs on t vertices?

Exercise 19

Consider running the **Contract** algorithm until the number of vertices is reduced to t, then use a cubic-time algorithm to find the min-cut in the contracted graph. Show that repeating this process sufficiently many times to ensure a success probability of $1/2$ requires $\Omega(n^{8/3})$ time.

We consider the following algorithm, that aims at improving the success probability:

Algorithm FastCut:

Input: A multigraph G.

Output: A cut C.

1. Let n be the number of vertices of G.
2. If $n \leq 6$, then compute the minimum cut by brute force. Otherwise,
   (a) Let $t \leftarrow \lceil 1 + n/\sqrt{2} \rceil$.
   (b) Using algorithm **Contract**, perform two independent contraction sequences to obtain graphs H_1 and H_2, each with t vertices.
   (c) Recursively compute cuts in each of H_1 and H_2.
   (d) return the smaller of the two cuts.

Note that in Step 2b, the execution of the **Contract** algorithm is stopped as soon as only t vertices are left. We now consider the running time and success probability of **FastCut**.

Theorem 15

The running time of the **FastCut** algorithm is $O(n^2 \log n)$.

Proof. Assuming that the **Contract** algorithm runs in $O(n^2)$ time, an upper bound on the running of **FastCut** is given by

$$T(n) = 2 \cdot T\left(\lceil 1 + \frac{n}{\sqrt{2}} \rceil\right) + O(n^2).$$

The solution of this recurrence is $T(n) = O(n^2 \log n)$. □

While we only incur a penalty of a $\log n$ factor in the running time, the success probability is now considerably higher.

Theorem 16

The **FastCut** algorithm succeeds in finding a minimum cut with probability $\Omega(1/\log n)$.

Proof. Suppose a min-cut of value k has survived in a contracted multigraph H with t vertices, and let H_1 and H_2 be the two contracted multigraphs in which the *FastCut* algorithm will recurse. The invocation of the algorithm on H will succeed if the two following conditions are satisfied:

1. the min-cut survives at least one of the two contraction sequences to H_1 and H_2,
2. the recursive invocation of *FastCut* on this branch is successful.

From Proposition 9.1, the probability of surviving a contraction sequence that shrinks the number of vertices from t to $\lceil 1 + t/\sqrt{2} \rceil$ is at least

$$\frac{\lceil 1 + t/\sqrt{2} \rceil (\lceil 1 + t/\sqrt{2} \rceil - 1)}{t(t-1)} \geq \frac{1}{2}.$$

This addresses the first point above. Let $P(t)$ denote the probability that *FastCut* succeeds on a multigraph on t vertices. We now have:

$$P(t) \geq 1 - \left(1 - \frac{1}{2} P(\lceil 1 + t/\sqrt{2} \rceil)\right)^2.$$

Let $p(k)$ denote the probability that algorithm *FastCut* succeeds at the kth level of the recursion, with $k = \Theta(\log t)$. Then

$$\begin{aligned} p(k+1) &\geq 1 - \left(1 - \frac{1}{2} p(k)\right)^2 \\ &= 1 - (1 - p(k) + p(k)^2/4) \\ &= p(k) - p(k)^2/4. \end{aligned}$$

We now perform the change of variable (guess a simple form for $p(k)$):

$$p(k) = \frac{4}{q(k)+1}.$$

The recurrence now translates to one on $q(k)$:

$$\begin{aligned}
\frac{4}{q(k+1)+1} &= \frac{4}{q(k)+1} - \frac{4}{(q(k)+1)^2} \\
\frac{1}{q(k+1)+1} &= \frac{1}{q(k)+1} - \frac{1}{(q(k)+1)^2} \\
\frac{1}{q(k+1)+1} &= \frac{q(k)}{(q(k)+1)^2} \\
q(k+1)+1 &= \frac{(q(k)+1)^2}{q(k)} \\
q(k+1) &= q(k)+1+\frac{1}{q(k)}.
\end{aligned}$$

This shows, by unrolling, that $q(k) = k + O(\log k)$, hence that $p(k) = \Omega(1/k)$. Since $k = \Theta(\log t)$, we get that the success probability on a multigraph H with t vertices is $\Omega(1/\log t)$.

□

Chapter 10

Linear programming

We now give an example of application of randomized algorithm for the famous linear programming problem, in the case where the dimension, or the number of variables, is small, but the number of constraints is large.

See textbook: Motwani & Raghavan, Chapter 9.10

10.1 Linear programming in fixed dimension

The linear programming problem consists in finding the extremum of a linear function of d variables, subject to n constraints that are linear inequalities on these variables. We denote the variables by $x_1, x_2, \ldots, x_d$ and define a *linear program* as follows:

$$\min_x cx, \text{ s.t. } Ax \leq b,$$

for a matrix $A \in \mathbb{R}^{n \times d}$ of the coefficients of the linear constraints, an objective function with coefficients $c \in \mathbb{R}^d$, and a column vector $b \in \mathbb{R}^n$.

The constraints can be seen as defining halfspaces in $\mathbb{R}^d$, and the intersection of these halfspaces is the *feasibility region* of the linear program, hence the set of all *fasible solutions*, or points that satisfy all the constraints. In what follows, we will make the simplifying assumption that every vertex of the feasibility region is defined by exactly d constraints, hence the feasibility region is a *simple polytope*. This assumption can be lifted using standard perturbation techniques, that we will not detail here.

We will denote by H the set of constraints, and by $\mathcal{O}(S)$ the optimal solution defined by a subset $S \subseteq H$ of the constraints. A *basis* is a subset $B \subseteq H$ such that $\mathcal{O}(B) > -\infty$ and $\mathcal{O}(B') < \mathcal{O}(B)$ for any $B' \subset B$. The *basis $\mathcal{B}(H)$ of H* is a basis $B \subseteq H$ such that $\mathcal{O}(B) = \mathcal{O}(H)$. Our goal, when solving the linear program, is therefore to identify the basis $\mathcal{B}(H)$.

There is a huge literature on the linear programming problem. The classical algorithm for this problem is the *simplex method*, devised by Dantzig in the 1940s. While extremely useful in practice, most variants of this method have been proved to have exponential worst-case running times. The existence of a pivoting rule for the simplex algorithm that makes it run in polynomial time is an open problem.

We consider the case where the dimension d is a constant, or is small with respect to the number n of constraints. if d is constant, we can show that the linear programming problem can be solved in time $O(n)$.

Proposition 22

A linear program with n constraints and $d = O(1)$ variables can be solved in time $O(n)$.

10.2 Seidel's algorithm

Seidel's algorithm is a randomized *incremental* algorithm: We consider the constraints one by one in a random order, and update the solution accordingly. It is best described in the following recursive form.

1. If $|H| = d$ then return $\mathcal{B}(H) = H$
2. Pick a random constraint h
3. Recursively find $\mathcal{B}(H \setminus \{h\})$
4. If $\mathcal{B}(H \setminus \{h\})$ does not violate h, return $\mathcal{B}(H) = \mathcal{B}(H \setminus \{h\})$
5. Otherwise, project all the constraints in $H \setminus \{h\}$ onto h and recursively solve this new linear program in $d - 1$ dimensions

The correctness of the algorithm follows by observing that either h is redundant, and we output the correct basis in that case, or it is not. In the latter case, it must be the case that the optimal solution (vertex of the feasibility region) lies on the hyperplane bounding h. This is then an opportunity to decrease the dimension.

Exercise 20

1. Establish the following recurrence on the running time $T(n, d)$ of Seidel's algorithm, where $n = |H|$ is the number of constraints and d is the number of variables:

$$T(n, d) \leq T(n-1, d) + O(d) + \frac{d}{n}(O(dn) + T(n-1, d-1)),$$

for $n > 1$ and $d > 1$.

2. Guess that
$$T(n,d) = Cnf(d)$$
for some constant C and some function f of d, and establish a recurrence on $f(d)$.

3. Prove that
$$T(n,d) = O(nd!).$$

Solution of Exercise 20

1. Since there are at most d extreme constraints, the probability that the randomly picked constraint h violates the current solution is at most d/n. This explains the factor in the second part of the right-hand expression. The two recursive terms are self-explanatory. The first $O(d)$ term is the cost of checking if h violates the current solution. The $O(dn)$ term is the cost of projecting all constraints onto h.

2. Let us substitute the expression $T(n,d) = Cnf(d)$ in the previous recurrence:
$$C(n-1)f(d)+O(d)+\frac{d}{n}(O(dn)+C(n-1)f(d-1)) \leq Cnf(d)-Cf(d)+Cdf(d-1)+O(d^2).$$
If we let C be the largest constant in all O terms, for $T(n,d) \leq Cnf(d)$, we must have the following condition on the function f:
$$-Cf(d) + Cdf(d-1) + Cd^2 \leq 0 \Leftrightarrow f(d) \geq d^2 + df(d-1).$$

3. let us fix $f(d) = d^2 + df(d-1)$, with $f(1) = 1$. We solve the recurrence on f by unrolling:
$$\begin{aligned}
f(d) &= d^2 + df(d-1) \\
&= d^2 + d[(d-1)^2 + (d-1)f(d-2)] \\
&= d^2 + d[(d-1)^2 + (d-1)[(d-2)^2 + (d-2)f(d-3)]] \\
&= d^2 + d[(d-1)^2 + (d-1)[(d-2)^2 + (d-2)[(d-3)^2 + (d-3)f(d-4)]]] \\
(\ldots) &= d^2 + d(d-1)^2 + d(d-1)(d-2)^2 + d(d-1)(d-2)(d-3)^2 + \ldots + \\
&\quad d(d-1)(d-2)\ldots\cdot 2\cdot 1 \\
&= d!\left(\frac{d^2}{d!} + \frac{(d-1)^2}{(d-1)!} + \frac{(d-2)^2}{(d-2)!}\ldots\right) \\
&\leq d!\cdot\sum_{i\geq 1}\frac{i^2}{i!} \\
&= O(d!).
\end{aligned}$$

Theorem 17

Seidel's algorithm runs in time $O(nd!)$.

Chapter 11

The secretary problem

The secretary problem is a classical probabilistic puzzle belonging to the family of so-called *stopping problems*, or *online selection problems*: We have a sequence of candidates, and we have to make the decision, at each step, of either selecting the current candidate, or moving on in the sequence, without any possibility of recourse.

11.1 A stopping problem

Suppose we wish to hire a new assistant among n candidates. The candidates are interviewed in turn, and are assigned a score after each interview. The scores assigned to each candidates are distinct. They are interviewed in random order, so that every permutation of the n candidates has the same probability $1/n!$.

The rule for hiring is special: the job must be offered immediately after the interview, otherwise the chance to hire the candidate is lost forever. We are seeking a suitable algorithm in order to maximize the probability of hiring the best candidate.

We restrict to a special family of algorithms. Given a parameter m, you interview the first m candidates, and do not hire anyone. Then you hire the first candidate that has a better score than all previous candidates. It can be shown that the optimal algorithm actually belongs to this family.

11.2 Optimal stopping

Let us denote by E_i the event that the ith candidate is the best and is hired. Let us further denote by E the event that the best candidate is hired. We can directly see that

$$P(E) = \sum_{i=1}^{n} P(E_i).$$

We also know that the probability that the ith candidate is the best is $1/n$, since the

permutation is random. Let us now consider the probability that the ith candidate is hired, given that it is the best. We make the following observation:

Proposition 23

For $i > m$, and conditioned on the event that the ith candidate is the best, the event *"the ith candidate is hired"* is the same as *"the maximum score among the first $i-1$ candidates is among the first m candidates"*.

Indeed, if the maximum is among the first m, then the ith candidate must be picked, since it is the overall maximum. On the other hand, if the maximum among the first $i-1$ is not among the first m, then the ith candidate will certainly not be picked.

Now observe that the probability that the the maximum score among the first $i-1$ candidates is among the first m candidates is equal to $m/(i-1)$. Overall, we obtain

$$P(\text{the } i\text{th candidate is hired} \mid \text{the } i\text{th candidate is the best}) = \begin{cases} 0 & \text{if } i \leq m, \\ \frac{m}{i-1} & \text{otherwise.} \end{cases}$$

To get the value of $P(E_i)$, we multiply this by the probability $1/n$ that the ith candidate is the best. From the previous sum, we get that:

$$P(E) = \sum_{i=1}^{n} P(E_i) = \frac{m}{n} \sum_{j=m+1}^{n} \frac{1}{j-1} \simeq \frac{m}{n} (\ln n - \ln m).$$

The derivative of this function with respect to m is

$$\frac{1}{n}(\ln n - \ln m) + \frac{m}{n}\left(-\frac{1}{m}\right) = \frac{1}{n}(\ln n - \ln m - 1),$$

which cancels at $\ln m = \ln n - 1 \Leftrightarrow m = n/e$. Hence the best such algorithm uses $m = n/e$ (more precisely, a close integer), so that the probability $P(E)$ of hiring the best candidate is roughly equal to the constant $1/e$.

Proposition 24

The best algorithm for the secretary problem is the following: Interview the first $\sim n/e$ candidates, and from then on, hire the first candidate that is better than all previous candidates.

Because $1/e \simeq 0.368$, this algorithm is often referred to as "the 37% rule".