Notes on Randomized Algorithms
CS 469/569: Fall 2014

James Aspnes

2014-12-17 20:04

Contents

List of Figures

List of Tables

List of Algorithms

Preface

These are notes for the Fall 2014 semester version of the Yale course CPSC 469/569 Randomized Algorithms. This document also incorporates the lecture schedule and assignments, as well as some sample assignments from previous semesters. Because this is a work in progress, it will be updated frequently over the course of the semester.

Notes from previous versions of the course can be found at `http://www.cs.yale.edu/homes/aspnes/classes/469/notes-2013.pdf`, `http://www.cs.yale.edu/homes/aspnes/classes/469/notes-2011.pdf`, and `http://www.cs.yale.edu/homes/aspnes/pinewiki/CS469(2f)2009.html`. Some sample assignments from these semesters can also be found in the appendix.

Much of the structure of the course follows the textbook, Mitzenmacher and Upfals's *Probability and Computing: Randomized Algorithms and Probabilistic Analysis* [MU05], with some material from Motwani and Raghavan's *Randomized Algorithms* [MR95]. In most cases you'll find these textbooks contain much more detail than what is presented here, so it is probably better to consider this document a supplement to them than to treat it as your primary source of information.

I would like to thank my students and teaching fellows for their help in pointing out numerous errors and omissions in earlier drafts of these notes.

Syllabus

Description

A study of randomized algorithms from several areas: graph algorithms, algorithms in algebra, approximate counting, probabilistically checkable proofs, and matrix algorithms. Topics include an introduction to tools from probability theory, including some inequalities such as Chernoff bounds.

Meeting times

Monday and Wednesday 11:35–12:50 in DL 220.

On-line course information

The lecture schedule, course notes, and all assignments can be found in a single gigantic PDF file at `http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf`. You should probably bookmark this file, as it will be updated frequently.

For office hours, see `http://www.cs.yale.edu/homes/aspnes#calendar`.

Staff

The instructor for the course is James Aspnes. Office: AKW 401. Email: `james.aspnes@gmail.com`. URL: `http://www.cs.yale.edu/homes/aspnes/`.

Textbook

The textbook for the class is: Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*.

Cambridge University Press, 2005. ISBN 0521835402. QA274 M574X 2005.

Reserved books at Bass library

These are other textbooks on randomized algorithms:

- Rajeev Motwani and Prabhakar Raghavan, *Randomized Algorithms*. Cambridge University Press, 1995. ISBN 0521474655. QA274 M68X 1995. Also available at `http://www.books24x7.com/marc.asp?isbn=0521474655` from Yale campus IP addresses.

  The classic textbook in the field.

- Juraj Hromkovič, *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms*. Springer, 2005. ISBN 9783540239499. QA274 .H76X 2005 (LC). Also available at `http://dx.doi.org/10.1007/3-540-27903-2` from Yale campus IP addresses.

  Intended to be a gentler introduction to randomized algorithms than Motwani and Raghavan, but not as comprehensive.

These are general references on probability theory:

- William Feller, *An Introduction to Probability Theory and Its Applications*, volumes 1 and 2. Wiley, 1968 (volume 1, 3rd edition); Wiley 1971 (volume 2, 2nd edition). QA273 F43 1968.

  The probability theory analog of Knuth's *Art of Computer Programming*: comprehensive, multiple volumes, every theoretical computer scientist of the right generation owns a copy. Volume 1, which covers discrete probability, is the most useful for computer science.

- Geoffrey R. Grimmett and David R. Stirzaker, *Probability and Random Processes*. Oxford University Press, 2001. ISBN 0198572220. QA273 G74X 2001.

  Similar in scope to Feller. A good alternative if you are on a budget.

Course requirements

Six homework assignments (60% of the semester grade) plus a final exam (40%).

Use of outside help

Students are free to discuss homework problems and course material with each other, and to consult with the instructor or a TA. Solutions handed in, however, should be the student's own work. If a student benefits substantially from hints or solutions received from fellow students or from outside sources, then the student should hand in their solution but acknowledge the outside sources, and we will apportion credit accordingly. Using outside resources in solving a problem is acceptable but plagiarism is not.

Clarifications for homework assignments

From time to time, ambiguities and errors may creep into homework assignments. Questions about the interpretation of homework assignments should be sent to the instructor at `james.aspnes@gmail.com`. Clarifications will appear in an updated version of the assignment.

Late assignments

Late assignments will not be accepted without a Dean's Excuse.

Lecture schedule

As always, the future is uncertain, so you should take parts of the schedule that haven't happened yet with a grain of salt. Readings refer to chapters or sections in the course notes, except for those specified as in MU or MR, which refer to the course textbook [MU05] or to the supplemental text [MR95].

Office hours, lecture times, and assignment due dates can be found at `http://www.cs.yale.edu/homes/aspnes#calendar`.

2014-08-27 Randomized algorithms. What they are and what we will do with them. Crossing the Quinnipiac River as an illustration between the difference between worst-case analysis, average-cases analysis, and worst-case randomized analysis. Two ways to compute the expected running time of QuickSort. A randomized algorithm for testing equality of polynomials. Las Vegas vs. Monte Carlo algorithms. Readings: Chapter 1; MU §1.1.

2014-08-29 Probability theory as used in analyzing randomized algorithms. Readings: Chapter 2 except §2.3.1.2; MU §1.2.

2014-09-03 Karger's min-cut algorithm. Random variables and expectations. Readings: §2.3.1.2, Chapter 3 through §3.4.1.1; MU §§1.4, 2.1-2.2.

2014-09-08 Expectation of a product: correlation and covariance. Conditional expectation and applications. Readings: rest of Chapter 3; rest of MU Chapter 2.

2014-09-10 Basic probabilistic inequalities: The union bound, Markov's inequality, Jensen's inequality. Preview of Chebyshev's inequality and variance. Readings: Chapter 4; MU §§3.1–3.2.

2014-09-15 More on Chebyshev's inequality and variance. Applications of Chebyshev's inequality: the second-moment method. Readings: §5.1; MU §§3.3–3.4.

2014-09-17 Chernoff bounds and applications. Valiant's randomized routing algorithm for the hypercube. Readings: §section-chernoff-bounds; MU §§4.1–4.2, 4.5.1.

2014-09-22 Analysis of Valiant's algorithm. Concentration bounds for sums of bounded independent random variables and martingales with bounded increments: Azuma-Hoeffding inequality, Readings: §5.3 through §5.3.2; MU §§12.1, 12.4.

2014-09-24 The UCB1 algorithm for multi-armed bandits. Method of bounded differences, McDiarmid's inequality, and applications. Readings: Rest of §5.3; [ACBF02], MU §12.5.

2014-09-29 Randomized search trees and treaps. Readings: Chapter 6 through §6.3; MR §§8.1–8.2, [SA96].

2014-10-01 Skip lists; start of hash tables. Readings: §6.4, Chapter 7 through §7.2.1; MR §8.3, MU §§5.5.1 and 13.3.1.

2014-10-06 More hash tables: tabulation hashing, FKS, start of cuckoo hashing. Readings §§7.2.2–7.4; [FKS84, PR04].

2014-10-08 Analysis of Cuckoo hashing. Basic structure of Bloom filters. Readings: §7.6 up to §7.6.2; MU §5.5.3.

2014-10-13 No lecture

2014-10-15 No lecture

2014-10-17 Make-up lecture at usual time and place. More Bloom filters: Analysis of Bloom filters, comparison to optimal space, counting Bloom filters, count-min sketches. Readings: rest of §7.6; MU §13.4.

2014-10-20 Martingales and stopping times. Readings: Chapter 8; MU §§12.2–12.3.

2014-10-27 Start of Markov chains: basic definitions, classification of states, stationary distributions and the ergodic theorem. Readings: §§9.1–9.2; MU §§7.1–7.2.

2014-10-29 Reversible Markov chains, the Metropolis-Hastings algorithm, the coupling lemma. Readings: §9.3, §9.4 through §9.4.2; MU §§7.4, 11.1.

2014-11-03 Better proof of the coupling lemma. More examples of coupling arguments. Readings: §§9.4.2–9.4.5; MU §11.2.

2014-11-05 Path coupling. In class we did the definition and the example of sampling graph colorings for $k \geq 2\Delta + 1$; the readings give applications to other problems that might also be enlightening. Readings: §9.4.6; MU §11.6.

2014-11-10 Markov chain convergence via spectral methods. Conductance and Cheeger's inequality. Readings: §§9.5–9.5.5; MR §11.3.

2014-11-12 Bounding conductance using canonical paths. Readings: §§9.5.6–9.5.8, rest of MR §11.3.

2014-11-17 Approximate counting: sampling lemma, Karp-Luby algorithm for #DNF, Dyer's algorithm for #KNAPSACK. Readings: Chapter 10; MU §§10.1–10.3, [Dye03].

2014-11-19 The probabilistic method: randomized constructions and existence proofs, approximate MAX CUT, randomized rounding for MAX SAT. Readings: §§11.1.2, 11.2.1, and 11.2.2; MU §§6.1–6.2.

2014-12-01 Derandomization: Adleman's theorem, pairwise independence, method of conditional probabilities. Readings: Chapter 12; MR §§2.3, MU §§6.3 and 13.1.2.

2014-12-03 Quantum computing. Readings: Chapter 13.

2014-12-15 Final exam, starting at 14:00, in 17 Hillhouse Avenue room 101. It will be a closed-book test covering all material discussed during the semester.

Chapter 1

Randomized algorithms

A randomized algorithm flips coins during its execution to determine what to do next. When considering a randomized algorithm, we usually care about its **expected worst-case** performance, which is the average amount of time it takes on the worst input of a given size. This average is computed over all the possible outcomes of the coin flips during the execution of the algorithm. We may also ask for a **high-probability bound**, showing that the algorithm doesn't consume too much resources most of the time.

In studying randomized algorithms, we consider pretty much the same issues as for deterministic algorithms: how to design a good randomized algorithm, and how to prove that it works within given time or error bounds. The main difference is that it is often easier to design a randomized algorithm—randomness turns out to be a good substitute for cleverness more often than one might expect—but harder to analyze it. So much of what one does is develop good techniques for analyzing the often very complex random processes that arise in the execution of an algorithm. Fortunately, in doing so we can often use techniques already developed by probabilists and statisticians for analyzing less overtly algorithmic processes.

Formally, we think of a randomized algorithm as a machine M that computes $M(x, r)$, where x is the problem input and r is the sequence of random bits. Our machine model is the usual **random-access machine** or **RAM** model, where we have a memory space that is typically polynomial in the size of the input n, and in constant time we can read a memory location, write a memory location, or perform arithmetic operations on integers of up to $O(\log n)$ bits.[1] In this model, we may find it easier to think of the

[1] This model is unrealistic in several ways: the assumption that we can perform arithmetic on $O(\log n)$-bit quantities in constant omits at least a factor of $\Omega(\log \log n)$ for

random bits as supplied as needed by some subroutine, where generating a random integer of size $O(\log n)$ takes constant time; the justification for this assumption is that it takes constant time to read the next $O(\log n)$-sized value from the random input.

Because the number of these various constant-time operations, and thus the running time for the algorithm as a whole, may depend on the random bits, it is now a **random variable**—a function on points in some probability space. The probability space Ω consists of all possible sequences r, each of which is assigned a probability $\Pr[r]$ (typically $2^{-|r|}$), and the running time for M on some input x is generally given as an **expected value**[2] $\mathrm{E}_r[\mathrm{time}(M(x,r))]$, where for any X,

$$\mathrm{E}_r[X] = \sum_{r\in\Omega} X(r)\Pr[r]\,. \tag{1.0.1}$$

We can now quote the performance of M in terms of this expected value: where we would say that a deterministic algorithms runs in time $O(f(n))$, where $n = |x|$ is the size of the input, we instead say that our randomized algorithm runs in **expected time** $O(f(n))$, which means that $\mathrm{E}_r[\mathrm{time}(M(x,r))] = O(f(|x|))$ for all inputs x.

This is distinct from traditional **worst-case analysis**, where there is no r and no expectation, and **average-case analysis**, where there is again no r and the value reported is not a maximum but an expectation over some distribution on x. The following trivial example shows the distinction.

1.1 A trivial example

Let us consider a variant of the classic card game **Find the Lady**[3] Here a dealer puts down three cards and we want to find a specific card among the three (say, the Queen of Spades). In this version, the dealer will let us turn over as many cards as we want, but each card we turn over will cost us a dollar. If we find the queen, we get two dollars.

---

addition and probably more for multiplication in any realistic implementation; while the assumption that we can address n^c distinct locations in memory in anything less than $n^{c/3}$ time in the worst case requires exceeding the speed of light. But for reasonably small n, this gives a pretty good approximation of the performance of real computers, which do in fact perform arithmetic and access memory in a fixed amount of time, although with fixed bounds on the size of both arithmetic operands and memory.

[2] We'll see more details of these and other concepts from probability theory in Chapters 2 and 3.

[3] Often called **Three-card Monte**, but that will get confusing when we talk about *Monte Carlo* algorithms later.

Because this is a toy example, we assume that the dealer is not cheating. The author is not responsible for the results of applying the analysis below to real-world games where this assumption does not hold.

A deterministic algorithm tries the cards in some fixed order. A clever dealer will place the Queen in the last place we look: so the worst-case payoff is a loss of a dollar.

In the average case, if we assume that all three positions are equally likely to hold the target card, then we turn over one card a third of the time, two cards a third of the time, and all three cards a third of the time; this gives an expected payoff of

$$\frac{1}{3}(1+2+3)-2=0.$$

But this requires making assumptions about the distribution of the cards, and we are no longer doing worst-case analysis.

The trick to randomized algorithms is that we can can obtain the same expected payoff even in the worst case by supplying the randomness ourselves. If we turn over cards in a random order, then the same analysis for the average case tells us we get the same expected payoff of 0—but unlike the average case, we get this expected performance no matter where the dealer places the cards.

1.2 Verifying polynomial identities

A less trivial example is described in [MU05, §1.1]. Here we are given two products of polynomials and we want to determine if they compute the same function. For example, we might have

$$\begin{aligned} p(x) &= (x-7)(x-3)(x-1)(x+2)(2\,x+5) \\ q(x) &= 2\,x^5 - 13\,x^4 - 21\,x^3 + 127\,x^2 + 121\,x - 210 \end{aligned}$$

These expressions both represent degree-5 polynomials, and it is not obvious without multiplying out the factors of p whether they are equal or not. Multiplying out all the factors of p may take as much as $O(d^2)$ time if we assume integer multiplication takes unit time and do it the straightforward way.[4] We can do better than this using randomization.

The trick is that evaluating $p(x)$ and $q(x)$ takes only $O(d)$ integer operations, and we will find $p(x) = q(x)$ only if either (a) $p(x)$ and $q(x)$ are really

[4] It can be faster if we do something sneaky like use fast Fourier transforms [SS71].

the same polynomial, or (b) x is a **root** of $p(x) - q(x)$. Since $p(x) - q(x)$ has degree at most d, it can't have more than d roots. So if we choose x uniformly at random from some much larger space, it's likely that we will not get a root. Indeed, evaluating $p(11) = 112320$ and $q(11) = 120306$ quickly shows that p and q are not in fact the same.

This is an example of a **Monte Carlo algorithm**, which is an algorithm that runs in a fixed amount of time but only gives the right answer some of the time. (In this case, with probability $1 - d/r$, where r is the size of the range of random integers we choose x from.) Monte Carlo algorithms have the unnerving property of not indicating when their results are incorrect, but we can make the probability of error as small as we like by running the algorithm repeatedly. For this particular algorithm, the probability of error after k trials is only $(d/r)^k$, which means that for fixed d/r we need $O(\log(1/\epsilon))$ iterations to get the error bound down to any given ϵ. If we are really paranoid, we could get the error down to 0 by testing $d + 1$ distinct values, but now the cost is as high as multiplying out p again.

The error for this algorithm is one-sided: if we find a **witness** to the fact that $p \neq q$, we are done, but if we don't, then all we know is that we haven't found a witness yet. We also have the property that if we check enough possible witnesses, we are guaranteed to find one.

A similar property holds in the classic **Miller-Rabin primality test**, a randomized algorithm for determining whether a large integer is prime or not.[5] The original version, due to Gary Miller [Mil76] showed that, as in polynomial identity testing, it might be sufficient to pick a particular set of deterministic candidate witnesses. Unfortunately, this result depends on the truth of the extended Riemann hypothesis, a notoriously difficult open problem in number theory. Michael Rabin [Rab80] demonstrated that choosing random witnesses was enough, if we were willing to accept a small probability of incorrectly identifying a composite number as prime.

For many years it was open whether it was possible to test primality deterministically in polynomial time without unproven number-theoretic assumptions, and the randomized Miller-Rabin algorithm was one of the most widely-used randomized algorithms for which no good deterministic alternative was known. Finally, Agrawal *et al.* [AKS04] demonstrated how to test primality deterministically using a different technique, although the cost of their algorithm is high enough that Miller-Rabin is still used in practice.

[5] We will not describe this algorithm here.

1.3 Randomized QuickSort

The **QuickSort** algorithm [Hoa61a] works as follows. For simplicity, we assume that no two elements of the array being sorted are equal.

- If the array has > 1 elements,
  - Pick a **pivot** p uniformly at random from the elements of the array.
  - Split the array into A_1 and A_2, where A_1 contains all elements $< p$ elements $> p$.
  - Sort A_1 and A_2 recursively and return the sequence A_1, p, A_2.
- Otherwise return the array.

The splitting step takes exactly $n - 1$ comparisons, since we have to check each non-pivot against the pivot. We assume all other costs are dominated by the cost of comparisons. How many comparisons does randomized QuickSort do on average?

There are two ways to solve this problem: the dumb way and the smart way. We'll do it the dumb way now and save the smart way for §1.3.2.

1.3.1 Brute force method: solve the recurrence

Let $T(n)$ be the expected number of comparisons done on an array of n elements. We have $T(0) = T(1) = 0$ and for larger n,

$$T(n) = \frac{1}{n} \sum_{k=0}^{n-1} \left(T(k) + T(n-1-k) \right). \tag{1.3.1}$$

Why? Because there are n equally-likely choices for our pivot (hence the $1/n$), and for each choice the expected cost is $T(k) + T(n-1-k)$, where k is the number of elements that land in A_1. Formally, we are using here the **law of total probability**, which says that for any random variable X and partition of the probability space into events $B_1 \dots B_n$, then

$$\mathrm{E}[X] = \sum B_i \, \mathrm{E}[X \mid B_i],$$

where

$$\mathrm{E}[X \mid B_i] = \frac{1}{\Pr[B_i]} \sum_{\omega \in B_i} X(\omega)$$

is the **conditional expectation** of X conditioned on B_i, which we can think of as just the average value of X if we know that B_i occurred. (See §2.3.1 for more details.)

So now we just have to solve this ugly recurrence. We can reasonably guess that when $n \geq 1, T(n) \leq an \log n$ for some constant a. Clearly this holds for $n = 1$. Now apply induction on larger n to get

$$\begin{aligned}
T(n) &= (n-1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-1-k)) \\
&= (n-1) + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \\
&= (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} T(k) \\
&\leq (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} ak\log k \\
&\leq (n-1) + \frac{2}{n}\int_{k=1}^{n} ak\log k \\
&= (n-1) + \frac{2a}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{4} + \frac{1}{4}\right) \\
&= (n-1) + \frac{2a}{n}\left(\frac{n^2\log n}{2} - \frac{n^2}{4} + \frac{1}{4}\right) \\
&= (n-1) + an\log n - \frac{an}{2} + \frac{a}{2n}.
\end{aligned}$$

If we squint carefully at this recurrence for a while we notice that setting $a = 2$ makes this less than or equal to $an \log n$, since the remaining terms become $(n-1) - n + 1/n = 1/n - 1$, which is negative for $n \geq 1$. We can thus confidently conclude that $T(n) \leq 2n \log n$ (for $n \geq 1$).

1.3.2 Clever method: use linearity of expectation

Alternatively, we can use **linearity of expectation** (which we'll discuss further in §3.4.1 to compute the expected number of comparisons used by randomized QuickSort.

Imagine we use the following method for choosing pivots: we generate a random permutation of all the elements in the array, and when asked to sort some subarray A', we use as pivot the first element of A' that appears in our list. Since each element is equally likely to be first, this is equivalent to the

actual algorithm. Pretend that we are always sorting the numbers $1 \dots n$ and define for each pair of elements $i < j$ the **indicator variable** X_{ij} to be 1 if i is compared to j at some point during the execution of the algorithm and 0 otherwise. Amazingly, we can actually compute the probability of this event (and thus $\mathrm{E}[X_{ij}]$): the only time i and j are compared is if one of them is chosen as a pivot before they are split up into different arrays. How do they get split up into different arrays? If some intermediate element k is chosen as pivot first, i.e., if some k with $i < k < j$ appears in the permutation before both i and j. Occurrences of other elements don't affect the outcome, so we can concentrate on the restriction of the permutations to just the numbers i through j, and we win if this restricted permutation starts with either i or j. This event occurs with probability $2/(j-i+1)$, so we have $\mathrm{E}[X_{ij}] = 2/(j-i+1)$. Summing over all pairs $i < j$ gives:

$$\begin{aligned}
\mathrm{E}\left[\sum_{i<j} X_{ij}\right] &= \sum_{i<j} \mathrm{E}[X_{ij}] \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{i=2}^{n} \sum_{k=2}^{i} \frac{2}{k} \\
&= \sum_{k=2}^{n} \frac{2(n-k+1)}{k} \\
&= \sum_{k=2}^{n} \left(\frac{2(n+1)}{k} - 2\right) \\
&= \sum_{k=2}^{n} \frac{2(n+1)}{k} - 2(n-1) \\
&= 2(n+1)(H_n - 1) - 2(n-1) \\
&= 2(n+1)H_n - 4n.
\end{aligned}$$

Here $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the n-th **harmonic number**, equal to $\ln n + \gamma + O(n^{-1})$, where $\gamma \approx 0.5772$ is the **Euler-Mascheroni constant** (whose exact value is unknown!). For asymptotic purposes we only need $H_n = \Theta(\log n)$.

For the first step we are taking advantage of the fact that linearity of

expectation doesn't care about the variables not being independent. The rest is just algebra.

This is pretty close to the bound of $2n \log n$ we computed using the recurrence in §1.3.1. Given that we now know the exact answer, we could in principle go back and use it to solve the recurrence exactly.[6]

Which way is better? Solving the recurrence requires less probabilistic **handwaving** (a more polite term might be "insight") but more grinding out inequalities, which is a pretty common trade-off. Since I am personally not very clever I would try the brute-force approach first. But it's worth knowing about better methods so you can try them in other situations.

1.4 Classifying randomized algorithms by their goals

1.4.1 Las Vegas vs Monte Carlo

One difference between QuickSort and polynomial equality testing that QuickSort always succeeds, but may run for longer than you expect; while the polynomial equality tester always runs in a fixed amount of time, but may produce the wrong answer. These are examples of two classes of randomized algorithms, which were originally named by László Babai [Bab79]:[7]

- A **Las Vegas algorithm** fails with some probability, but we can tell when it fails. In particular, we can run it again until it succeeds, which means that we can eventually succeed with probability 1 (but with a potentially unbounded running time). Alternatively, we can think of a Las Vegas algorithm as an algorithm that runs for an unpredictable amount of time but always succeeds (we can convert such an algorithm back into one that runs in bounded time by declaring that it fails if it runs too long—a condition we can detect). QuickSort is an example of a Las Vegas algorithm.

- A **Monte Carlo algorithm** fails with some probability, but we can't tell when it fails. If the algorithm produces a yes/no answer and the failure probability is significantly less than 1/2, we can reduce the probability of failure by running it many times and taking a majority of the answers. The polynomial equality-testing algorithm in an example of a Monte Carlo algorithm.

[6] We won't.

[7] To be more precise, Babai defined *Monte Carlo algorithms* based on the properties of **Monte Carlo simulation**, a technique dating back to the Manhattan project. The term *Las Vegas algorithm* was new.

The heuristic for remembering which class is which is that the names were chosen to appeal to English speakers: in Las Vegas, the dealer can tell you whether you've won or lost, but in Monte Carlo, *le croupier ne parle que Français*, so you have no idea what he's saying.

Generally, we prefer Las Vegas algorithms, because we like knowing when we have succeeded. But sometimes we have to settle for Monte Carlo algorithms, which can still be useful if we can get the probability of failure small enough. For example, any time we try to estimate an average by **sampling** (say, inputs to a function we are trying to integrate or political views of voters we are trying to win over) we are running a Monte Carlo algorithm: there is always some possibility that our sample is badly non-representative, but we can't tell if we got a bad sample unless we already know the answer we are looking for.

1.4.2 Randomized complexity classes

Las Vegas vs Monte Carlo is the typical distinction made by algorithm designers, but complexity theorists have developed more elaborate classifications. These include algorithms with "one-sided" failure properties. For these algorithms, we never get a bogus "yes" answer but may get a bogus "no" answer (or vice versa). This gives us several complexity classes that act like randomized versions of **NP**, co-**NP**, etc.:

- The class **R** or **RP** (randomized **P**) consists of all languages L for which a polynomial-time Turing machine M exists such that if $x \in L$, then $\Pr[M(x,r)=1] \geq 1/2$ and if $x \notin L$, then $\Pr[M(x,r)=1] = 0$. In other words, we can find a witness that $x \in L$ with constant probability. This is the randomized analog of **NP** (but it's much more practical, since with **NP** the probability of finding a winning witness may be exponentially small).

- The class co-**R** consists of all languages L for which a poly-time Turing machine M exists such that if $x \notin L$, then $\Pr[M(x,r)=1] \geq 1/2$ and if $x \in L$, then $\Pr[M(x,r)=1] = 0$. This is the randomized analog of co-**NP**.

- The class **ZPP** (zero-error probabilistic P) is defined as **RP** $\cap$ co-**RP**. If we run both our **RP** and co-**RP** machines for polynomial time, we learn the correct classification of x with probability at least $1/2$. The rest of the time we learn only that we've failed (because both machines return 0, telling us nothing). This is the class of (polynomial-time)

Las Vegas algorithms. The reason it is called "zero-error" is that we can equivalently define it as the problems solvable by machines that always output the correct answer eventually, but only run in *expected* polynomial time.

- The class **BPP** (bounded-error probabilistic **P**) consists of all languages L for which a poly-time Turing machine exists such that if $x \notin L$, then $\Pr[M(x,r)=1] \leq 1/3$, and if $x \in L$, then $\Pr[M(x,r)=1] \geq 2/3$. These are the (polynomial-time) Monte Carlo algorithms: if our machine answers 0 or 1, we can guess whether $x \in L$ or not, but we can't be sure.

- The class **PP** (probabilistic **P**) consists of all languages L for which a poly-time Turing machine exists such that if $x \notin L$, then $\Pr[M(x,r)=1] \geq 1/2$, and if $x \in L$, then $\Pr[M(x,r)=1] < 1/2$. Since there is only an exponentially small gap between the two probabilities, such algorithms are not really useful in practice; **PP** is mostly of interest to complexity theorists.

Assuming we have a source of random bits, any algorithm in **RP**, co-**RP**, **ZPP**, or **BPP** is good enough for practical use. We can usually even get away with using a pseudorandom number generator, and there are good reasons to suspect that in fact every one of these classes is equal to **P**.

1.5 Classifying randomized algorithms by their methods

We can also classify randomized algorithms by how they use their randomness to solve a problem. Some very broad categories:[8]

- **Avoiding worst-case inputs**, by hiding the details of the algorithm from the adversary. Typically we assume that an adversary supplies our input. If the adversary can see what our algorithm is going to do (for example, he knows which door we will open first), he can use this information against us. By using randomness, we can replace our predictable deterministic algorithm by what is effectively a random choice of many different deterministic algorithms. Since the adversary doesn't know which algorithm we are using, he can't (we hope) pick an input that is bad for all of them.

[8] These are largely adapted from the introduction to [MR95].

- **Sampling**. Here we use randomness to find an example or examples of objects that are likely to be typical of the population they are drawn from, either to estimate some average value (pretty much the basis of all of statistics) or because a typical element is useful in our algorithm (for example, when picking the pivot in QuickSort). Randomization means that the adversary can't direct us to non-representative samples.

- **Hashing**. Hashing is the process of assigning a large object x a small name $h(x)$ by feeding it to a **hash function** h. Because the names are small, the Pigeonhole Principle implies that many large objects hash to the same name (a **collision**). If we have few objects that we actually care about, we can avoid collisions by choosing a hash function that happens to map them to different places. Randomization helps here by keeping the adversary from choosing the objects after seeing what our hash function is.

  Hashing techniques are used both in **load balancing** (e.g., insuring that most cells in a **hash table** hold only a few objects) and in **fingerprinting** (e.g., using a **cryptographic hash function** to record a **fingerprint** of a file, so that we can detect when it has been modified).

- **Building random structures**. The **probabilistic method** shows the existence of structures with some desired property (often graphs with interesting properties, but there are other places where it can be used) by showing that a randomly-generated structure in some class has a nonzero probability of having the property we want. If we can beef the probability up to something substantial, we get a randomized algorithm for generating these structures.

- **Symmetry breaking**. In **distributed algorithms** involving multiple processes, progress may be stymied by all the processes trying to do the same thing at the same time (this is an obstacle, for example, in **leader election**, where we want only one process to declare itself the leader). Randomization can break these deadlocks.

Chapter 2

Probability theory

In this chapter, we summarize the parts of **probability theory** that we need for the course. This is not really a substitute for reading an actual probability theory book like Feller [Fel68] or Grimmett and Stirzaker [GS01].

The basic idea of probability theory is that we want to model all possible outcomes of whatever process we are studying simultaneously. This gives the notion of a **probability space**, which is the set of all possible outcomes; for example, if we roll two dice, the probability space would consist of all 36 possible combinations of values. Subsets of this space are called **events**; an example in the two-dice space would be the event that the sum of the two dice is 11, given by the set $A = \{\langle 5, 6\rangle, \langle 6, 5\rangle\}$. The probability of an event A is given by a **probability measure** $\Pr[A]$; for simple probability spaces, this is just the sum of the probabilities of the individual outcomes contains in A, while for more general spaces, we define the measure on events first and the probabilities of individual outcomes are derived from this. Formal definitions of all of these concepts are given later in this chapter.

When analyzing a randomized algorithm, the probability space describes all choices of the random bits used by the algorithm, and we can think of the possible executions of an algorithm as living within this probability space. More formally, the sequence of operations carried out by an algorithm and the output it ultimately produces are examples of **random variables**—functions from a probability space to some other set—which we will discuss in detail in Chapter 3.

2.1 Probability spaces and events

A **discrete probability space** is a countable set Ω of **points** or **outcomes** ω. Each ω in Ω has a **probability** $\Pr[\omega]$, which is a real value with $0 \leq \Pr[\omega] \leq 1$. It is required that $\sum_{\omega \in \Omega} = 1$.

An **event** A is a subset of Ω; its probability is $\Pr[A] = \sum_{\omega \in A} \Pr[\omega]$. We require that $\Pr[\Omega] = 1$, and it is immediate from the definition that $\Pr[\emptyset] = 0$.

The **complement** $\bar{A}$ or $\neg A$ of an event A is the event $\Omega - A$. It is always the case that $\Pr[\neg A] = 1 - \Pr[A]$.

This fact is a special case of the general principle that if $A_1, A_2, \ldots$ forms a **partition** of Ω—that is, if $A_i \cap A_j = \emptyset$ when $i \neq j$ and $\bigcup A_i = \Omega$—then $\sum \Pr[A_i] = 1$. It happens to be the case that $\neg A$ and A form a partition of Ω consisting of exactly two elements.

Whenever $A_1, A_2, \ldots$ are disjoint events (i.e., when $A_i \cap A_j = \emptyset$ for all $i \neq j$), it is the case that $\Pr[\bigcup A_i] = \sum \Pr[A_i]$. This fact does not hold in general for events that are not disjoint.

For discrete probability spaces, all of these facts can be proven directly from the definition of probabilities for events. For more general probability spaces, it's no longer possible to express the probability of an event as the sum of the probabilities of its elements, and we adopt an axiomatic approach instead.

2.1.1 General probability spaces

More general probability spaces consist of a triple $(\Omega, \mathcal{F}, \Pr)$ where Ω is a set of points, $\mathcal{F}$ is a **σ-algebra** (a family of subsets of Ω that contains Ω and is closed under complement and countable unions) of **measurable sets**, and Pr is a function from $\mathcal{F}$ to $[0, 1]$ that gives $\Pr[\Omega] = 1$ and satisfies **countable additivity**: when $A_1, \ldots$ are disjoint, $\Pr[\bigcup A_i] = \sum \Pr[A_i]$. This definition is needed for uncountable spaces, because (under certain set-theoretic assumptions) we may not be able to assign a meaningful probability to all subsets of Ω.

Formally, this definition is often presented as three **axioms of probability**, due to Kolmogorov [Kol33]:

1. $\Pr[A] \geq 0$ for all $A \in \mathcal{F}$.

2. $\Pr[\Omega] = 1$.

3. For any countable collection of disjoint events $A_1, A_2, \ldots,$

$$\Pr\left[\bigcup_i A_i\right] = \sum_i \Pr\left[A_i\right].$$

It's not hard to see that the discrete probability spaces defined in the preceding section satisfy these axioms.

General probability spaces arise in randomized algorithms when we have an algorithm that might consume an unbounded number of random bits. The problem now is that an outcome consists of countable sequence of bits, and there are uncountably many such outcomes. The solution is to consider as measurable events only those sets with the property that membership in them can be determined after a finite amount of time. Formally, the probability space Ω the set $\{0,1\}^{\mathbb{N}}$ of all sequences of 0 and 1 values indexed by the natural numbers, and the measurable sets $\mathcal{F}$ are all sets that can be generated by countable unions and intersections of **cylinder sets**, where a cylinder set consists of all extensions xy of some finite prefix x. The probability measure itself is obtained by assigning the set of all points that start with x the probability $2^{-|x|}$, and computing the probabilities of other sets from the axioms. [1]

An oddity that arises in general probability spaces is it may be that every particular outcome has probability zero but their union has probability 1. For example, the probability of any particular infinite string of bits is 0, but the set containing all such strings is the entire space and has probability 1. This is where the fact that probabilities only add over *countable* unions comes in.

Most randomized algorithms books gloss over general probability spaces, with three good reasons. The first is that if we truncate an algorithm after a finite number of steps, we are usually get back to a discrete probability space, which avoids a lot of worrying about measurability and convergence. The second is that we are often implicitly working in a probability space that is either discrete or well-understood (like the space of bit-vectors described above). The last is that the **Kolmogorov extension theorem** says that if we specify $\Pr\left[A_1 \cap A_2 \cap \cdots \cap A_k\right]$ consistently for all finite sets of events $\{A_1 \ldots A_k\}$, then there exists some probability space that makes

[1] This turns out to give the same probabilities as if we consider each outcome as a real number in the interval $[0,1]$ and use Lebesgue measure to compute the probability of events. For some applications, thinking of our random values as real numbers (or even sequences of real numbers) can make things easier: consider for example what happens when we want to choose one of three outcomes with equal probability.

these probabilities work, even if we have uncountably many such events. So it's usually enough to specify how the events we care about interact, without worrying about the details of the underlying space.

2.2 Boolean combinations of events

Even though events are defined as sets, we often think of them as representing propositions that we can combine using the usual Boolean operations of NOT ($\neg$), AND ($\wedge$), and OR ($\vee$). In terms of sets, these correspond to taking a complement $\bar{A} = \Omega \setminus A$, an intersection $A \cap B$, or a union $A \cup B$.

We can use the axioms of probability to calculate the probability of $\bar{A}$:

Lemma 2.2.1.

$$\Pr\left[\bar{A}\right] = 1 - \Pr\left[A\right].$$

Proof. First note that $A \cap \bar{A} = \emptyset$, so $A \cup \bar{A} = \Omega$ is a disjoint union of countably many[2] events. This gives $\Pr\left[A\right] + \Pr\left[\bar{A}\right] = \Pr\left[\Omega\right] = 1$. □

For example, if our probability space consists of the six outcomes of a fair die roll, and $A = [\text{outcome is } 3]$ with $\Pr\left[A\right] = 5/6$, then $\Pr\left[\text{outcome is not } 3\right] = \Pr\left[\bar{A}\right] = 1 - 1/6 = 5/6$. Though this example is trivial, using the formula does save us from having to add up the five cases where we don't get 3.

If we want to know the probability of $A \cap B$, we need to know more about the relationship between A and B. For example, it could be that A and B are both events representing a fair coin coming up heads, with $\Pr\left[A\right] = \Pr\left[B\right] = 1/2$. The probability of $A \cap B$ could be anywhere between $1/2$ and 0:

- For ordinary fair coins, we'd expect that half the time that A happens, B also happens. This gives $\Pr\left[A \cap B\right] = (1/2) \cdot (1/2) = 1/4$. To make this formal, we might define our probability space Ω as having four outcomes HH, HT, TH, and TT, each of which occurs with equal probability.

- But maybe A and B represent the same fair coin: then $A \cap B = A$ and $\Pr\left[A \cap B\right] = \Pr\left[A\right] = 1/2$.

- At the other extreme, maybe A and B represent two fair coins welded together so that if one comes up heads the other comes up tails. Now $\Pr\left[A \cap B\right] = 0$.

[2] Countable need not be infinite, so 2 is countable.

- With a little bit of tinkering, we could also find probabilities for the outcomes in our four-outcome space to make $\Pr[A] = \Pr[\mathsf{HH}] + \Pr[\mathsf{HT}] = 1/2$ and $\Pr[B] = \Pr[\mathsf{HH}] + \Pr[\mathsf{TH}] = 1/2$ while setting $\Pr[A \cap B] = \Pr[\mathsf{HH}]$ to any value between 0 and 1/2.

The difference between the nice case where $\Pr[A \cap B] = 1/4$ and the other, more annoying cases where it doesn't is that in the first case we have assumed that A and B are **independent**, which is *defined* to mean that $\Pr[A \cap B] = \Pr[A]\Pr[B]$.

In the real world, we expect events to be independent if they refer to parts of the universe that are not causally related: if we flip two coins that aren't glued together somehow, then we assume that the outcomes of the coins are independent. But we can also get independence from events that are not causally disconnected in this way. An example would be if we rolled a fair four-sided die labeled $\mathsf{HH}, \mathsf{HT}, \mathsf{TH}, \mathsf{TT}$, where we take the first letter as representing A and the second as B.

There's no simple formula for $\Pr[A \cup B]$ when A and B are not disjoint, even for independent events, but we can compute the probability by splitting up into smaller, disjoint events and using countable additivity:

$$\begin{aligned}
\Pr[A \cup B] &= \Pr\left[(A \cap B) \cup (A \cap \bar{B}) \cup (\bar{A} \cap B)\right] \\
&= \Pr[A \cap B] + \Pr\left[A \cap \bar{B}\right] + \Pr\left[\bar{A} \cap B\right] \\
&= \left(\Pr[A \cap B] + \Pr\left[A \cap \bar{B}\right]\right) + \left(\Pr\left[\bar{A} \cap B\right] + \Pr[A \cap B]\right) - \Pr[A \cap B] \\
&= \Pr[A] + \Pr[B] - \Pr[A \cap B].
\end{aligned}$$

The idea is that we can compute $\Pr[A \cup B]$ by adding up the individual probabilities and then subtracting off the part where the counted the event twice.

This is a special case of the general **inclusion-exclusion formula**, which says:

Lemma 2.2.2. *For any finite sequence of events $A_1 \ldots A_n$,*

$$\begin{aligned}
\Pr\left[\bigcup_{i=1}^{n} A_i\right] &= \sum_i \Pr[A_i] - \sum_{i<j} \Pr[A_i \cap A_j] + \sum_{i<j<k} \Pr[A_i \cap A_j \cap A_k] - \ldots \\
&= \sum_{S \subseteq \{1 \ldots n\}, S \neq \emptyset} (-1)^{|S|+1} \Pr\left[\bigcap_{i \in S} A_i\right]. \qquad (2.2.1)
\end{aligned}$$

Proof. Partition Ω into 2^n disjoint events B_T, where $B_T = (\bigcap_{i \in T} A_i) \cap \left(\bigcap_{i \notin T} \bar{A}_i\right)$ is the event that all A_i occur for i in T and no A_i occurs for i not in T. Then A_i is the union of all B_T with $T \ni i$ and $\bigcup A_i$ is the union of all B_T with $T \neq \emptyset$.

That the right-hand side gives the probability of this event is a sneaky consequence of the binomial theorem, and in particular the fact that $\sum_{i=1}^{n}(-1)^i = \sum_{i=0}^{n}(-1)^i - 1 = (1-1)^n - 1$ is zero if $n > 0$ and -1 if $n = 0$. Using this fact after rewriting the right-hand side using the B_T events gives

$$\begin{aligned}
\sum_{S \subseteq \{1 \ldots n\}, S \neq \emptyset} (-1)^{|S|+1} \Pr\left[\bigcap_{i \in S} A_i\right] &= \sum_{S \subseteq \{1 \ldots n\}, S \neq \emptyset} (-1)^{|S|+1} \sum_{T \supseteq S]} \Pr\left[B_T\right] \\
&= \sum_{T \subseteq \{1 \ldots n\}} \left(\Pr\left[B_T\right] \sum_{S \subseteq T, S \neq \emptyset} (-1)^{|S|+1}\right) \\
&= \sum_{T \subseteq \{1 \ldots n\}} \left(-\Pr\left[B_T\right] \sum_{i=1}^{n} (-1)^i \binom{|T|}{i}\right) \\
&= \sum_{T \subseteq \{1 \ldots n\}} \left(-\Pr\left[B_T\right] \left((1-1)^{|T|} - 1\right)\right) \\
&= \sum_{T \subseteq \{1 \ldots n\}} \Pr\left[B_T\right] \left(\left(1 - 0^{|T|}\right)\right) \\
&= \sum_{T \subseteq \{1 \ldots n\}, T \neq \emptyset} \Pr\left[B_T\right] \\
&= \Pr\left[\bigcap_{i=1}^{n} A_i\right].
\end{aligned}$$

□

2.3 Conditional probability

The **probability of** A **conditioned on** B or **probability of** A **given** B, written $\Pr[A \mid B]$, is defined by

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}, \tag{2.3.1}$$

provided $\Pr[B \neq 0]$. If $\Pr[B] = 0$, we generally can't condition on B.

Such conditional probabilities represent the effect of restricting our probability space to just B, which can think of as computing the probability of

each event if we know that B occurs. The intersection in the numerator limits A to circumstances where B occurs, while the denominator normalizes the probabilities so that, for example, $\Pr[\Omega \mid B] = \Pr[B \mid B] = 1$.

2.3.1 Application to algorithm analysis

The reason we like conditional probability in algorithm analysis is that it gives us a natural way to model the kind of case analysis that we are used to applying to deterministic algorithms. Suppose we are trying to prove that a randomized algorithm works (event A) with a certain probability. Most likely, the first random thing the algorithm does is flip a coin, giving two possible outcomes B and $\bar{B}$. Countable additivity tells us that $\Pr[A] = \Pr[A \cap B] + \Pr\left[A \cap \bar{B}\right]$, which we can rewrite using conditional probability as

$$\Pr[A] = \Pr[A \mid B]\Pr[B] + \Pr\left[A \;\middle|\; \bar{B}\right]\Pr\left[\bar{B}\right], \tag{2.3.2}$$

a special case of the **law of total probability**.

What's nice about this expression is that we can often compute $\Pr[A \mid B]$ and $\Pr\left[A \;\middle|\; \bar{B}\right]$ by looking at what the algorithm does starting from the point where it has just gotten heads (B) or tails ($\bar{B}$), and use the formula to combine these values to get the overall probability of success.

For example, if

$$\begin{aligned}
\Pr[\text{class occurs} \mid \text{snow}] &= 3/5, \\
\Pr[\text{class occurs} \mid \text{no snow}] &= 99/100, \text{and} \\
\Pr[\text{snow}] &= 1/10,
\end{aligned}$$

then

$$\Pr[\text{class occurs}] = (3/5)\cdot(1/10) + (99/100)\cdot(1 - 1/10) \quad = 0.951.$$

More generally, we can do the same computation for any partition of Ω into countably many disjoint events B_i:

$$\begin{aligned}
\Pr[A] &= \Pr\left[\bigcup_i (A \cap B_i)\right] \\
&= \sum_i \Pr[A \cap B_i] \\
&= \sum_{i, \Pr[B_i] \neq 0} \Pr[A \mid B_i]\Pr[B_i], \qquad (2.3.3)
\end{aligned}$$

which is the **law of total probability**. Note that the last step works for each term only if $\Pr[A \mid B_i]$ is well-defined, meaning that $\Pr[B_i] \neq 0$. But any such case contributes nothing to the previous sum, so we get the correct answer if we simply omit any terms from the sum for which $\Pr[B_i] = 0$.

A special case arises when $\Pr\left[A \,\middle|\, \bar{B}\right] = 0$, which occurs, for example, if $A \subseteq B$. Then we just have $\Pr[A] = \Pr[A \mid B]\Pr[B]$. If we consider an event $A = A_1 \cap A_2 \cap \cdots \cap A_k$, then we can iterate this expansion to get

$$\begin{aligned}
\Pr[A_1 \cap A_2 \cap \cdots \cap A_k] &= \Pr[A1 \cap \cdots \cap A_{k-1}]\Pr[A_k \mid A_1, \ldots, A_{k-1}] \\
&= \Pr[A1 \cap \cdots \cap A_{k-2}]\Pr[A_{k-1} \mid A_1, \ldots, A_{k-2}]\Pr[A_k \mid A_1, \ldots, A_{k-1}] \\
&= \ldots \\
&= \prod_{i=1}^{k} \Pr[A_i \mid A_1, \ldots, A_i]. \qquad (2.3.4)
\end{aligned}$$

Here $\Pr[A \mid B, C, \ldots]$ is short-hand for $\Pr[B \cap C \cap \ldots]$, the probability that A occurs given that all of B, C, etc., occur.

2.3.1.1 Racing coin-flips

Suppose that I flip coins and allocate a space for each heads that I get before the coin comes up tails. Suppose that you then supply me with objects (each of which takes up one unit of space), one for each heads that *you* get before you get tails. What are my chances of allocating enough space?

Let's start by solving this directly using the law of total probability. Let A_i be the event that I allocate i spaces. The event A_i is the intersection of i independent events that I get heads in the first i positions and the event that I get tails in position $i+1$; this multiplies out to $(1/2)^{i+1}$. Let B_i be the similar event that you supply i objects. Let W be the event that I win. To make the A_i partition the space, we must also add an extra event A_∞ equal to the singleton set $\{\mathsf{HHHHHHH}\ldots\}$ consisting of the all-H sequence; this has probability 0 (so it won't have much of an effect), but we need to include it since $\mathsf{HHHHHHH}\ldots$ is not contained in any of the other A_i.

We can compute

$$\begin{aligned}
\Pr[W \mid A_i] &= \Pr[B_0 \cap B_1 \cap \cdots \cap B_i \mid A_i] \\
&= \Pr[B_0 \cap B_1 \cap \cdots \cap B_i] \\
&= \Pr[B_0] + \Pr[B_1] + \cdots + \Pr[B_i] \\
&= \sum_{j=0}^{i} (1/2)^i \\
&= (1/2) \cdot \frac{1 - (1/2)^{i+1}}{1 - 1/2} \\
&= 1 - (1/2)^{i+1}. \qquad (2.3.5)
\end{aligned}$$

The clean form of this expression suggests strongly that there is a better way to get it, and that this way involves taking the negation of the intersection of $i+1$ independent events that occur with probability 1/2 each. With a little reflection, we can see that the probability that your objects *don't* fit in my buffer is exactly $(1/2)^{i+1}$

From the law of total probability (2.3.3),

$$\begin{aligned}
\Pr[W] &= \sum_{i=0}^{\infty} (1 - (1/2)^{i+1})(1/2)^{i+1} \\
&= 1 - \sum_{i=0}^{\infty} (1/4)^{i+1} \\
&= 1 - \frac{1}{4} \cdot 11 - 1/4 \\
&= 2/3.
\end{aligned}$$

This gives us our answer. However, we again see an answer that is suspiciously simple, which suggests looking for another way to find it. We can do this using conditional probability by defining new events C_i, where C_i contains all sequences of coin-flips for both players where get i heads in a row but at least one gets tails on the $(i+1)$-th coin. These events plus the probability-zero event $C_\infty = \{\mathsf{HHHHHHH}\ldots, \mathsf{TTTTTTT}\ldots\}$ partition the space, so $\Pr[W] = \sum_{i=0}^{\infty} \Pr[W \mid C_i] \Pr[C_i]$.

Now we ask, what is $\Pr[W \mid C_i]$? Here we only need to consider three cases, depending on the outcomes of our $(i+1)$-th coin-flips. The cases HT and TT cause me to win, while the case TH causes me to lose, and each occurs with equal probability conditioned on C_i (which excludes HH). So I win 2/3 of the time conditioned on C_i, and summing $\Pr[W] = \sum_{i=0}^{\infty} (2/3) \Pr[C_i] = 2/3$ since I know that $\Pr[C_i]$ sums to 1 from the axioms.

Still another approach is to compute the probability that our runs have exactly the same length ($\sum_{i=1}^{\infty} 2^{-i} \cdot 2^{-i} = 1/3$), and argue by symmetry that the remaining 2/3 probability is equally split between my run being longer (1/3) and your run being longer (1/3). Since W occurs if my run is just as long or longer, $\Pr[W] = 1/3 + 1/3 = 2/3$. A nice property of this approach is that the only summation involved is over disjoint events, so we get to avoid using conditional probability entirely.

2.3.1.2 Karger's min-cut algorithm

Here we'll give a simple algorithm for finding a global **min-cut** in a **multigraph**,[3] due to David Karger [Kar93].

The idea is that we are given a multigraph G, and we want to partition the vertices into nonempty sets S and T such that the number of edges with one endpoint in S and one endpoint in T is as small as possible. There are many efficient ways to do this, most of which are quite sophisticated. There is also the algorithm we will now present, which solves the problem with reasonable efficiency using almost no sophistication at all (at least in the algorithm itself).

The main idea is that given an edge uv, we can construct a new multigraph G_1 by **contracting** the edge: in G_1, u and v are replaced by a single vertex, and any edge that used to have either vertex as an endpoint now goes to the combined vertex (edges with both endpoints in $\{u, v\}$ are deleted). Karger's algorithm is to contract edges chosen uniformly at random until only two vertices remain. All the vertices that got packed into one of these become S, the others become T. It turns out that this finds a minimum cut with probability at least $1/\binom{n}{2}$.

Theorem 2.3.1. *Given any min cut (S, T) of a graph G on n vertices, Karger's algorithm outputs (S, T) with probability at least $1/\binom{n}{2}$.*

Proof. Let (S, T) be a min cut of size k. Then the degree of each vertex v is at least k (otherwise $(v, G - v)$ would be a smaller cut), and G contains at least $kn/2$ edges. The probability that we contract an S–T edge is thus at most $k/(kn/2) = 2/n$, and the probability that we don't contract one is $1 - 2/n = (n - 2)/n$. Assuming we missed collapsing (S, T) the first time, we now have a new graph G_1 with $n - 1$ vertices in which the min cut is still of size k. So now the chance that we miss (S, T) is $(n - 3)/(n - 1)$. We stop

[3] Unlike ordinary graphs, multigraphs can have more than one edge between two vertices.

when we have two vertices left, so the last step succeeds with probability 1/3.

We can compute the probability that the S–T cut is never contracted by multiplying all the condition probabilities together using (2.3.4)

$$\prod_{i=3}^{n} \frac{i-2}{i} = \frac{2}{n(n-1)}.$$

□

If the graph has more than one min cut, this only makes our life easier. Note that since each min cut turns up with probability at least $1/\binom{n}{2}$, there can't be more than $\binom{n}{2}$ of them.[4] But even if there is only one, we have a good chance of finding it if we simply re-run the algorithm substantially more than n^2 times.

2.3.2 Conditional probability and independence

Rearranging (2.3.1) gives $\Pr[A \cap B] = \Pr[B]\Pr[A \mid B] = \Pr[A]\Pr[B \mid A]$. In many cases, knowing that B occurs tells us nothing about whether A occurs; if so, we have $\Pr[A \mid B] = \Pr[B]$, which implies that $\Pr[A \cap B] = \Pr[A \mid B]\Pr[B] = \Pr[A]\Pr[B]$—events A and B are **independent**. So $\Pr[A \mid B] = \Pr[A]$ gives an alternative criterion for independence when $\Pr[B]$ is nonzero.[5]

A *set* of events $A_1, A_2, \ldots$ is independent if A_i is independent of B when B is any Boolean formula of the A_j for $j \neq i$. The idea is that you can't predict A_i by knowing anything about the rest of the events.

A set of events $A_1, A_2, \ldots$ is **pairwise independent** if each A_i and A_j, $i \neq j$ are independent. It is possible for a set of events to be pairwise independent but not independent; a simple example is when A_1 and A_2 are the events that two independent coins come up heads and A_3 is the event that both coins come up with the same value. The general version of pairwise independence is **k-wise independence**, which means that any subset of k (or fewer) events are independent.

[4] The suspiciously combinatorial appearance of the $1/\binom{n}{2}$ suggests that there should be some way of associating minimum cuts with particular pairs of vertices, but I'm not aware of any natural way to do this. It may be that sometimes the appearance of a simple expression in a surprising context may just stem from the fact that there aren't very many distinct simple expressions.

[5] If $\Pr[B]$ is zero, then A and B are always independent.

Chapter 3

Random variables

A **random variable** on a probability space Ω is just a function with domain Ω.[1] Rather than writing a random variable as $f(\omega)$ everywhere, the convention is to write a random variable as a capital letter (X, Y, S, etc.) and make the argument implicit: X is really $X(\omega)$. Variables that aren't random (or aren't variable) are written in lowercase.

For example, consider the probability space corresponding to rolling two independent fair six-sided dice. There are 36 possible outcomes in this space, corresponding to the 6×6 pairs of values $\langle x, y \rangle$ we might see on the two dice. We could represent the value of each die as a random variable X or Y given by $X(\langle x, y \rangle) = x$ or $Y(\langle x, y \rangle) = y$, but for many applications, we don't care so much about the specific values on each die. Instead, we want to know the sum $S = X + Y$ of the dice. This value S is also random variable; as a function on Ω, it's defined by $S(\langle x, y \rangle) = x + y$.

Random variables need not be real-valued. There's no reason why we can't think of the pair $\langle x, y \rangle$ itself a random variable, whose range is the set $[1 \ldots 6] \times [1 \ldots 6]$. Similarly, if we imagine choosing a point uniformly at random in the unit square $[0, 1]^2$, its coordinates are a random variable. For a more exotic example, the **random graph** $G_{n,p}$ obtained by starting with n vertices and including each possible edge with independent probability p is a random variable whose range is the set of all graphs on n vertices.

[1] Technically, this only works for discrete spaces. In general, a random variable is a **measurable function** from a probability space $(\Omega, \mathcal{F})$ to some other set S equipped with its own σ-algebra $\mathcal{F}'$. What makes a function measurable in this sense is that that for any set A in $\mathcal{F}'$, the inverse image $f^{-1}(A)$ must be in $\mathcal{F}$. See §3.3 for more details.

3.1 Operations on random variables

Random variables may be combined using standard arithmetic operators, have functions applied to them, etc., to get new random variables. For example, the random variable X/Y is a function from Ω that takes on the value $X(\omega)/Y(\omega)$ on each point ω.

3.2 Random variables and events

Any random variable X allows us to define events based on its possible values. Typically these are expressed by writing a predicate involving the random variable in square brackets. An example would be the probability that the sum of two dice is exactly 11: $[S = 11]$; or that the sum of the dice is less than 5: $[S < 5]$. These are both sets of outcomes; we could expand $[S = 11] = \{\langle 5, 6\rangle, \langle 6, 5\rangle\}$ or $[S < 5] = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 3, 1\rangle\}$. This allows us to calculate the probability that a random variable has particular properties: $\Pr[S = 11] = \frac{2}{36} = \frac{1}{18}$ and $\Pr[S < 5] = \frac{6}{36} = \frac{1}{6}$.

Conversely, given any event A, we can define an **indicator random variable** 1_A that is 1 when A occurs and 0 when it doesn't.[2] Formally, $1_A(\omega) = 1$ for ω in A and $1_A(\omega) = 0$ for ω not in A.

Indicator variables are mostly useful when combined with other random variables. For example, if you roll two dice and normally collect the sum of the values but get nothing if it is 7, we could write your payoff as $S \cdot 1_{[S \neq 7]}$.

The **probability mass function** of a random variable gives $\Pr[X = x]$ for each possible value x. For example, our random variable S has the probability mass function show in Table 3.1. For a discrete random variable X, the probability mass function gives enough information to calculate the probability of any event involving X, since we can just sum up cases using countable additivity. This gives us another way to compute $\Pr[S < 5] = \Pr[S = 2] + \Pr[S = 3] + \Pr[S = 4] = \frac{1+2+3}{36} = \frac{1}{6}$.

For two random variables, the **joint probability mass function** gives $\Pr[X = x \wedge Y = y]$ for each pair of values x and y (this generalizes in the obvious way for more than two variables).

[2] Some people like writing χ_A for these.

You may also see $[P]$ where P is some predicate, a convention known as **Iverson notation** or the **Iverson bracket** that was invented by Iverson for the programming language APL, appears in later languages like C where the convention is that true predicates evaluate to 1 and false ones to 0, and ultimately popularized for use in mathematics—with the specific choice of square brackets to set off the predicate—by Graham *et al.* [GKP88].

Out of these alternatives, I personally find 1_A to be the least confusing.

S	Probability
2	1/36
3	2/36
4	3/36
5	4/36
6	5/36
7	6/36
8	5/36
9	4/36
10	3/36
11	2/36
12	1/36

Table 3.1: Probability mass function for the sum of two independent fair six-sided dice

We will often refer to the probability mass function as giving the **distribution** or **joint distribution** of a random variable or collection of random variables, even though distribution (for real-valued variables) technically refers to the **cumulative distribution function** $F(x) = \Pr[X \leq x]$, which is generally not directly computable from the probability mass function for **continuous random variables** that take on uncountably many values. To the extent that we can, we will try to avoid continuous random variables, and the rather messy integration theory needed to handle them.

Two or more random variables are **independent** if all sets of events involving different random variables are independent. In terms of probability mass functions, X and Y are independent if $\Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$. In terms of cumulative distribution functions, X and Y are independent if $\Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$. As with events, we generally assume that random variables associated with causally disconnected processes are independent, but this is not the only way we might have independence.

It's not hard to see that the individual die values X and Y in our two-dice example are independent, because every possible combination of values x and y has the same probability $1/36 = \Pr[X = x] \Pr[Y = y]$. If we chose a different probability distribution on the space, we might not have independence.

3.3 Measurability

For discrete probability spaces, any function on outcomes can be a random variable. The reason is that any event in a discrete probability space has a well-defined probability. For more general spaces, in order to be useful, events involving a random variable should have well-defined probabilities. For **discrete random variables** that take on only countably many values (e.g., integers or rationals), it's enough for the event $[X = x]$ (that is, the set $\{\omega \mid X(\omega) = x\}$) to be in $\mathcal{F}$ for all x. For real-valued random variables, we ask that the event $[X \leq x]$ be in $\mathcal{F}$. In these cases, we say that X is **measurable** with respect to $\mathcal{F}$, or just **measurable** $\mathcal{F}$. More exotic random variables use a definition of measurability that generalizes the real-valued version, which we probably won't need.[3] Since we usually just assume that

[3] The general version is that if X takes on values on another measure space $(\Omega', \mathcal{F}')$, then the inverse image $X^{-1}(A) = \{\omega \in \Omega \mid X(\omega) \in A\}$ of any set A in $\mathcal{F}'$ is in $\mathcal{F}$. This means in particular that $\Pr_\Omega$ maps through X to give a probability measure on Ω' by $\Pr_{\Omega'}[A] = \Pr_\Omega[X^{-1}(A)]$, and the condition on $X^{-1}(A)$ being in $\mathcal{F}$ makes this work.

all of our random variables are measurable unless we are doing something funny with $\mathcal{F}$ to represent ignorance, this issue won't come up much.

3.4 Expectation

The **expectation** or **expected value** of a random variable X is given by $\mathrm{E}[X] = \sum_x x \Pr[X = x]$. This is essentially an average value of X weighted by probability, and it only makes sense if X takes on values that can be summed in this way (e.g., real or complex values, or vectors in a real- or complex-valued vector space). Even if the expectation makes sense, it may be that a particular random variable X doesn't have an expectation, because the sum fails to converge.

For example, if X and Y are independent fair six-sided dice, then $\mathrm{E}[X] = \mathrm{E}[Y] = \sum_{i=1}^{6} i \left(\frac{1}{6}\right) = \frac{21}{6} = \frac{7}{2}$, while $\mathrm{E}[X+Y]$ is the rather horrific

$$\begin{aligned}\sum_{i=2}^{12} i \Pr[X+Y=i] &= \frac{2\cdot 1 + 3\cdot 2 + 4\cdot 3 + 5\cdot 4 + 6\cdot 5 + 7\cdot 6 + 8\cdot 5 + 9\cdot 4 + 10\cdot 3 + 11\cdot 2 + 12\cdot 1}{36} \\ &= \frac{252}{36} = 7.\end{aligned}$$

The fact that $7 = \frac{7}{2} + \frac{7}{2}$ here is not a coincidence.

3.4.1 Linearity of expectation

The main reason we like expressing the run times of algorithms in terms of expectation is **linearity of expectation**: $\mathrm{E}[aX + bY] = \mathrm{E}[aX] + \mathrm{E}[bY]$ for all random variables X and Y for which $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ are defined, and all constants a and b. This means that we can compute the running time for different parts of our algorithm separately and then add them together, *even if the costs of different parts of the algorithm are not independent.*

The general version is $\mathrm{E}\left[\sum a_i X_i\right] = \sum a_i \mathrm{E}[X_i]$ for any *finite* collection of random variables X_i and constants a_i, which follows by applying induction to the two-variable case. A special case is $\mathrm{E}[cX] = c\,\mathrm{E}[X]$ when c is constant.

For discrete random variables, linearity of expectation follows immediately from the definition of expectation and the fact that the event $[X = x]$

is the disjoint union of the events $[X = x, Y = y]$ for all y:

$$\begin{aligned}
\mathrm{E}[aX + bY] &= \sum_{x,y} (ax + by) \Pr[X = x \wedge Y = y] \\
&= a \sum_{x,y} x \Pr[X = x, Y = y] + b \sum_{x,y} y \Pr[X = x, Y = y] \\
&= a \sum_{x} x \sum_{y} Pr[X = x, Y = y] + b \sum_{y} y \sum_{x} \Pr[X = x, Y = y] \\
&= a \sum_{x} x Pr[X = x] + b \sum_{y} y \Pr[Y = y] \\
&= a \mathrm{E}[X] + b \mathrm{E}[Y].
\end{aligned}$$

Note that this proof does *not* require that X and Y be independent. The sum of two fair six-sided dice always has expectation $\frac{7}{2} + \frac{7}{2} = 7$, whether they are independent dice, the same die counted twice, or one die X and its complement $7 - X$.

Linearity of expectation makes it easy to compute the expectations of random variables that are expressed as sums of other random variables. One example that will come up a lot is a **binomial random variable**, which is the sum $S = \sum_{i=1}^{n} X_i$, where each X_i is an independent, identically distributed random variable that is 1 with probability p and 0 otherwise (such a random variable is called a **Bernoulli random variable**.[4] In this case each X_i individually has $\mathrm{E}[X_i] = p$, so $\mathrm{E}[S]$ is just np.

3.4.1.1 Linearity of expectation for infinite sequences

For infinite sequences of random variables, linearity of expectation may break down. This is true even if the sequence is countable. An example is the **St. Petersburg paradox**, in which a gambler bets \$1 on a double-or-nothing game, then bets \$2 if she loses, then \$4, and so on, until she eventually wins and stops, up \$1. If we represent the gambler's gain or loss at stage i as a random variable X_i, it's easy to show that $\mathrm{E}[X_i] = 0$, because the gambler either wins $\pm 2^i$ with equal probability, or doesn't play at all. So $\sum_{i=0}^{\infty} \mathrm{E}[X_i] = 0$. But $\mathrm{E}[\sum_{i=0}^{\infty} X_i] = 1$, because the probability that the gambler doesn't eventually win is zero.[5]

[4] Named for , the 18th-century Swiss mathematician.

[5] The trick here is that we are trading a probability-1 gain of 1 against a probability-0 loss of ∞. So we could declare that $\mathrm{E}\left[\sum_{i=0}^{\infty} X_i\right]$ involves $0 \cdot (-\infty)$ and is undefined. But this would lose the useful property that expectation isn't affected by probability-0 outcomes. As often happens in mathematics, we are forced to choose between candidate

Fortunately, these pathological cases don't come up often in algorithm analysis, and with some additional side constraints we can apply linearity of expectation even to infinite sums of random variables. The simplest is when $X_i \geq 0$ for all i; then $\mathrm{E}\left[\sum_{i=0}^{\infty} X_i\right]$ exists and is equal to $\sum_{i=0}^{\infty} \mathrm{E}\left[X_i\right]$ whenever the sum of the expectations converges (this is a consequence of the monotone convergence theorem). Another condition that works is if $\left|\sum_{i=0}^{n} X_i\right| \leq Y$ for all n, where Y is a random variable with finite expectation; the simplest version of this is when Y is constant. See [GS92, §5.6.12] or [Fel71, §IV.2]for more details.

3.4.2 Expectation of a product

When two random variables X and Y are independent, it also holds that $\mathrm{E}\left[XY\right] = \mathrm{E}\left[X\right]\mathrm{E}\left[Y\right]$. The proof (at least for discrete random variables) is straightforward:

$$\begin{aligned}
\mathrm{E}\left[XY\right] &= \sum_x \sum_y xy \Pr\left[X = x, Y = y\right] \\
&= \sum_x \sum_y xy \Pr\left[X = x\right] \Pr\left[Y = y\right] \\
&= \left(\sum_x x \Pr\left[X = x\right]\right) \left(\sum_y \Pr\left[Y = y\right]\right) \\
&= \mathrm{E}\left[X\right]\mathrm{E}\left[Y\right].
\end{aligned}$$

For example, the expectation of the product of two independent fair six-sided dice is $\left(\frac{7}{2}\right)^2 = \frac{49}{4}$.

This is not true for arbitrary random variables. If we compute the expectation of the product of a single fair six-sided die with itself, we get $\frac{1\cdot 1+2\cdot 2+3\cdot 3+4\cdot 4+5\cdot 5+6\cdot 6}{6} = \frac{91}{6}$ which is much larger.

The difference $\mathrm{E}\left[XY\right] - \mathrm{E}\left[X\right]\cdot\mathrm{E}\left[Y\right]$ is called the **covariance** of X and Y, written $\mathrm{Cov}\left[X, Y\right]$. It will come back later when we look at concentration bounds in Chapter 5.

3.4.2.1 Wald's equation (simple version)

Computing the expectation of a product does not often come up directly in the analysis of a randomized algorithm. Where we might expect to do it is

definitions based on which bad consequences we most want to avoid, with no way to avoid all of them. So the standard definition of expectation allows the St. Petersburg paradox because the alternatives are worse.

when we have a loop: one random variable N tells us the number of times we execute the loop, while another random variable X tells us the cost of each iteration. The problem is that if each iteration is randomized, then we really have a sequence of random variables $X_1, X_2, \ldots$, and what we want to calculate is

$$\mathrm{E}\left[\sum_{i=1}^{N} X_i\right], \tag{3.4.1}$$

where we can't use the sum formula directly because N is a random variable and we can't use the product formula because the X_i are all different random variables.

If N and the X_i are all independent, and N is bounded by some fixed maximum n, then we can apply the product rule to get the value of (3.4.1) by throwing in a few indicator variables. The idea is that the contribution of X_i to the sum is given by $X_i 1_{[N \geq i]}$, and because we assume that N is independent of the X_i, if we need to compute $\mathrm{E}\left[X_i 1_{[N \geq i]}\right]$, we can do so by computing $\mathrm{E}\left[X_i\right] \mathrm{E}\left[1_{[N \geq i]}\right]$.

So we get

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=1}^{N} X_i\right] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i 1_{[N \geq i]}\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[X_i 1_{[N \geq i]}\right] \\
&= \sum_{i=1}^{n} \mathrm{E}\left[X_i\right] \mathrm{E}\left[1_{[N \geq i]}\right].
\end{aligned}$$

For general X_i we have to stop here. But if we also know that the X_i all have the same expectation μ, then $\mathrm{E}\left[X_i\right]$ doesn't depend on i and we can bring it out of the sum. This gives

$$\begin{aligned}
\sum_{i=1}^{n} \mathrm{E}\left[X_i\right] \mathrm{E}\left[1_{[N \geq i]}\right] &= \mu \sum_{i=1}^{n} \mathrm{E}\left[1_{[N \geq i]}\right] \\
&= \mu\, \mathrm{E}\left[N\right]. \qquad (3.4.2)
\end{aligned}$$

This equation is a special case of **Wald's equation**, which we will see again in §8.4.2. The main difference between this version and the general version is that here we had to assume that N was independent of the X_i, which may not be true if our loop is a `while` loop, and termination is correlated with the time taken by the last iteration.

But for simple cases, (3.4.2) can still be useful. For example, if we throw one six-sided die to get N, and then throw N six-sided dice and add them up, we get the same expected total $\frac{7}{2} \cdot \frac{7}{2} = \frac{49}{4}$ as if we just multiply two six-sided dice. This is true even though the actual distribution of values is very different in the two cases.

3.5 Conditional expectation

We can also define a notion of **conditional expectation**, analogous to conditional probability. There are three versions of this, depending on how fancy we want to get about specifying what information we are conditioning on:

- The conditional expectation of X conditioned on an *event* A is written $\mathrm{E}[X \mid A]$ and defined by $\mathrm{E}[X \mid A] = \sum_x x \Pr[X = x \mid A] = \sum_x \frac{\Pr[X=x \wedge A]}{\Pr[A]}$. This is essentially the weighted average value of X if we know that A occurs.

- The expectation of X conditioned on a *random variable* Y, written $\mathrm{E}[X \mid Y]$, is a random variable. Assuming Y is discrete, we define $\mathrm{E}[X \mid Y]$ by the rule that for each $\omega \in \Omega$, $\mathrm{E}[X \mid Y](\omega) = \mathrm{E}[X \mid Y = Y(\omega)]$.[6] The intuition behind this definition is that $\mathrm{E}[X \mid Y]$ is a weighted-average estimate of X given that we know the value of Y but nothing else. Similar, we can define $\mathrm{E}[X \mid Y_1, Y_2, \ldots]$ to be the expected value of X given that we know the values of $Y_1, Y_2, \ldots$.

  One way to think about this is to use indicator variables to sum together a bunch of expectations conditioned on events. When Y is discrete, we can write

$$\mathrm{E}[X \mid Y] = \sum_y 1_{[Y=y]} \cdot \mathrm{E}[X \mid Y = y]. \tag{3.5.1}$$

- Expectation conditioned on a random variable is actually a special case of the expectation of X conditioned on a *σ-algebra* $\mathcal{F}$. Recall that a σ-algebra is a family of subsets of Ω that includes Ω and is closed under complement and countable union; for discrete probability spaces, this turns out to be the set of all unions of equivalence classes

[6] If Y is not discrete, the situation is more complicated. See [Fel71, §§III.2 and V.9–V.11].

for some equivalence relation on Ω,[7] and we think of $\mathcal{F}$ as representing knowledge of which equivalence class we are in, but not which point in the equivalence class we land on. An example would be if Ω consists of all values (X_1, X_2) obtained from two die rolls, and $\mathcal{F}$ consists of all sets A such that whenever one point ω with $X_1(\omega) + X_2(\omega) = s$ is in A, so is every other point ω' with $X_1(\omega') + X_2(\omega') = s$. (This is the σ-algebra **generated by** the random variable $X_1 + X_2$.)

A discrete random variable X is **measurable** $\mathcal{F}$ if every event $[X = x]$ is contained in $\mathcal{F}$; in other words, knowing only where we are in $\mathcal{F}$, we can compute exactly the value of X.

If X is not measurable $\mathcal{F}$, the best approximation we can make to it given that we only know where we are in $\mathcal{F}$ is $\mathrm{E}[X \mid \mathcal{F}]$, which is defined as a random variable Q that is (a) measurable $\mathcal{F}$; and (b) satisfies $\mathrm{E}[Q \mid A] = \mathrm{E}[X \mid A]$ for any non-null $A \in \mathcal{F}$. For discrete probability spaces, this just means that we replace X with its average value across each equivalence class: property (a) is satisfied because $\mathrm{E}[X \mid \mathcal{F}]$ is constant across each equivalence class, meaning that $[\mathrm{E}[X \mid \mathcal{F}] = x]$ is a union of equivalence classes, and property (b) is satisfied because we define $\mathrm{E}[\mathrm{E}[X \mid \mathcal{F}] \mid A] = \mathrm{E}[X \mid A]$ for each equivalence class A, and the same holds for unions of equivalence classes by a simple calculation.

3.5.1 Examples

- Let X be the value of a six-sided die. Let A be the event that X is even. Then

$$\begin{aligned}\mathrm{E}[X \mid A] &= \sum_x x \Pr[X = x \mid A] \\ &= (2 + 4 + 6) \cdot \frac{1}{3} \\ &= 4.\end{aligned}$$

- Let X and Y be independent six-sided dice, and let $Z = X + Y$. Then $\mathrm{E}[Z \mid X]$ is a random variable whose value is $1 + 7/2$ when $X = 1$,

[7] Proof: Let $\mathcal{F}$ be a σ-algebra over a countable set Ω. Let $\omega \sim \omega'$ if, for all A in $\mathcal{F}$, $\omega \in A$ if and only if $\omega' \in A$; this is an equivalence relation on Ω. To show that the equivalence classes of $\sim$ are elements of $\mathcal{F}$, for each $\omega'' \not\sim \omega$, let $A_{\omega''}$ be some element of $\mathcal{F}$ that contains ω but not ω''. Then $\bigcap_{\omega''} A_{\omega''}$ (a countable intersection of elements of $\mathcal{F}$) contains ω and all points $\omega' \sim \omega$ but no points $\omega'' \not\sim \omega$; in other words, it's the equivalence class of ω. Since there are only countably many such equivalence classes, we can construct all the elements of $\mathcal{F}$ by taking all possible unions of them.

$2 + 7/2$ when $X = 2$, etc. We can write this succinctly by writing $\mathrm{E}[Z \mid X] = X + 7/2$.

- Conversely, if X, Y, and Z are as above, we can also compute $\mathrm{E}[X]\, Z$. Here we are told what Z is and must make an estimate of X.

  For some values of Z, this nails down X completely: $\mathrm{E}[X \mid Z = 2] = 1$ because X you can only make 2 in this model as $1 + 1$. For other values, we don't know much about X, but can still compute the expectation. For example, to compute $\mathrm{E}[X \mid Z = 5]$, we have to average X over all the pairs (X, Y) that sum to 5. This gives $\mathrm{E}[X \mid Z = 5] = \frac{1}{4}(1 + 2 + 3 + 4) = \frac{5}{2}$. (This is not terribly surprising, since by symmetry $\mathrm{E}[Y \mid Z = 5]$ should equal $\mathrm{E}[X \mid Z = 5]$, and since conditional expectations add just like regular expectations, we'd expect that the sum of these two expectations would be 5.)

  The actual random variable $\mathrm{E}[X \mid Z]$ summarizes these conditional expectations for all events of the form $[Z = z]$. Because of the symmetry argument above, we can write it succinctly as $\mathrm{E}[X \mid Z] = \frac{Z}{2}$. Or we could list its value for every ω in our underlying probability space, as in done in Table 3.2 for this and various other conditional expectations on the two-independent-dice space.

3.5.2 Properties of conditional expectations

Most properties of expectations also hold for conditional expectations. For example, $\mathrm{E}[X + Y \mid Z] = \mathrm{E}[X \mid Z] + \mathrm{E}[Y \mid Z]$and $\mathrm{E}[cX \mid Y] = c\,\mathrm{E}[X \mid Y]$ when c is constant. But there are some additional properties that are often useful.

One is that computing the expectation of a condition expectation yields the expectation: $\mathrm{E}[\mathrm{E}[X \mid Y]] = \mathrm{E}[X]$. This is known as the **law of total expectation** or **law of iterated expectation**, and is the expectation version of the law of total probability.

This is often used in reverse, to introduce a conditional expectation and let us compute $\mathrm{E}[X]$ piecewise.

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}[\mathrm{E}[X \mid Y]] \\ &= \sum_y \mathrm{E}[X \mid Y = y]\,\Pr[Y = y]\,.\end{aligned}$$

The law of total expectation also works with partial conditioning: $\mathrm{E}[\mathrm{E}[X \mid Y, Z] \mid Y] = \mathrm{E}[X \mid Y]$.

ω	X	Y	$Z = X + Y$	$\mathrm{E}[X]$	$\mathrm{E}[X \mid Y = 3]$	$\mathrm{E}[X \mid Y]$	$\mathrm{E}[Z \mid X]$	$\mathrm{E}[X \mid Z]$	$\mathrm{E}[X \mid X]$
$(1,1)$	1	1	2	$7/2$	$7/2$	$7/2$	$1+7/2$	$2/2$	1
$(1,2)$	1	2	3	$7/2$	$7/2$	$7/2$	$1+7/2$	$3/2$	1
$(1,3)$	1	3	4	$7/2$	$7/2$	$7/2$	$1+7/2$	$4/2$	1
$(1,4)$	1	4	5	$7/2$	$7/2$	$7/2$	$1+7/2$	$5/2$	1
$(1,5)$	1	5	6	$7/2$	$7/2$	$7/2$	$1+7/2$	$6/2$	1
$(1,6)$	1	6	7	$7/2$	$7/2$	$7/2$	$1+7/2$	$7/2$	1
$(2,1)$	2	1	3	$7/2$	$7/2$	$7/2$	$2+7/2$	$3/2$	2
$(2,2)$	2	2	4	$7/2$	$7/2$	$7/2$	$2+7/2$	$4/2$	2
$(2,3)$	2	3	5	$7/2$	$7/2$	$7/2$	$2+7/2$	$5/2$	2
$(2,4)$	2	4	6	$7/2$	$7/2$	$7/2$	$2+7/2$	$6/2$	2
$(2,5)$	2	5	7	$7/2$	$7/2$	$7/2$	$2+7/2$	$7/2$	2
$(2,6)$	2	6	8	$7/2$	$7/2$	$7/2$	$2+7/2$	$8/2$	2
$(3,1)$	3	1	4	$7/2$	$7/2$	$7/2$	$3+7/2$	$4/2$	3
$(3,2)$	3	2	5	$7/2$	$7/2$	$7/2$	$3+7/2$	$5/2$	3
$(3,3)$	3	3	6	$7/2$	$7/2$	$7/2$	$3+7/2$	$6/2$	3
$(3,4)$	3	4	7	$7/2$	$7/2$	$7/2$	$3+7/2$	$7/2$	3
$(3,5)$	3	5	8	$7/2$	$7/2$	$7/2$	$3+7/2$	$8/2$	3
$(3,6)$	3	6	9	$7/2$	$7/2$	$7/2$	$3+7/2$	$9/2$	3
$(4,1)$	4	1	5	$7/2$	$7/2$	$7/2$	$4+7/2$	$5/2$	4
$(4,2)$	4	2	6	$7/2$	$7/2$	$7/2$	$4+7/2$	$6/2$	4
$(4,3)$	4	3	7	$7/2$	$7/2$	$7/2$	$4+7/2$	$7/2$	4
$(4,4)$	4	4	8	$7/2$	$7/2$	$7/2$	$4+7/2$	$8/2$	4
$(4,5)$	4	5	9	$7/2$	$7/2$	$7/2$	$4+7/2$	$9/2$	4
$(4,6)$	4	6	10	$7/2$	$7/2$	$7/2$	$4+7/2$	$10/2$	4
$(5,1)$	5	1	6	$7/2$	$7/2$	$7/2$	$5+7/2$	$6/2$	5
$(5,2)$	5	2	7	$7/2$	$7/2$	$7/2$	$5+7/2$	$7/2$	5
$(5,3)$	5	3	8	$7/2$	$7/2$	$7/2$	$5+7/2$	$8/2$	5
$(5,4)$	5	4	9	$7/2$	$7/2$	$7/2$	$5+7/2$	$9/2$	5
$(5,5)$	5	5	10	$7/2$	$7/2$	$7/2$	$5+7/2$	$10/2$	5
$(5,6)$	5	6	11	$7/2$	$7/2$	$7/2$	$5+7/2$	$11/2$	5
$(6,1)$	6	1	7	$7/2$	$7/2$	$7/2$	$6+7/2$	$7/2$	6
$(6,2)$	6	2	8	$7/2$	$7/2$	$7/2$	$6+7/2$	$8/2$	6
$(6,3)$	6	3	9	$7/2$	$7/2$	$7/2$	$6+7/2$	$9/2$	6
$(6,4)$	6	4	10	$7/2$	$7/2$	$7/2$	$6+7/2$	$10/2$	6
$(6,5)$	6	5	11	$7/2$	$7/2$	$7/2$	$6+7/2$	$11/2$	6
$(6,6)$	6	6	12	$7/2$	$7/2$	$7/2$	$6+7/2$	$12/2$	6

Table 3.2: Various conditional expectations on two independent dice

A useful generalization of linearity of expectation is that when Z is a function of Y, then $\mathrm{E}[XZ \mid Y] = Z\,\mathrm{E}[X \mid Y]$. Essentially, Z acts like a constant over each event $[Y = y]$, so we can pull it out.

3.6 Applications

3.6.1 Geometric random variables

Suppose that we are running a Las Vegas algorithm that takes a fixed amount of time T, but succeeds only with probability p (which we take to be independent of the outcome of any other run of the algorithm). If the algorithm fails, we run it again. How long does it take on average to get the algorithm to work?

We can reduce the problem to computing $\mathrm{E}[TX] = T\,\mathrm{E}[X]$, where X is the number of times the algorithm runs. The probability that $X = n$ is exactly $(1-p)^{n-1}p$, because we need to get $n-1$ failures with probability $1-p$ each followed by a single success with probability p, and by assumption all of these probabilities are independent. A variable with this kind of distribution is called a **geometric random variable**. We saw a special case of this distribution earlier (§2.3.1.1) when we were looking at how long it took to get a tails out of a fair coin (in that case, p was 1/2).

Using conditional expectation, it's straightforward to compute $\mathrm{E}[X]$. Let A be the event that the algorithm succeeds on the first run, i.e., then event $[X = 1]$. Then

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}[X \mid A]\,\Pr[A] + \mathrm{E}\left[X \;\middle|\; \bar{A}\right]\Pr\left[\bar{A}\right] \\ &= 1 \cdot p + \mathrm{E}\left[X \;\middle|\; \bar{A}\right] \cdot (1-p).\end{aligned}$$

The tricky part here is to evaluate $\mathrm{E}\left[X \;\middle|\; \bar{A}\right]$. Intuitively, if we don't succeed the first time, we've wasted one step and are back where we started, so it should be the case that $\mathrm{E}\left[X \;\middle|\; \bar{A}\right] = 1 + \mathrm{E}[X]$. If we want to be really careful, we can calculate this out formally (no sensible person would ever do

this):

$$\begin{aligned}
\mathrm{E}\left[X \;\middle|\; \bar{A}\right] &= \sum_{n=1}^{\infty} n \Pr\left[X = n \mid X \neq 1\right] \\
&= \sum_{n=2}^{\infty} n \frac{\Pr\left[X = n\right]}{\Pr\left[X \neq 1\right]} \\
&= \sum_{n=2}^{\infty} n \frac{(1-p)^{n-1}p}{1-p} \\
&= \sum_{n=2}^{\infty} n(1-p)^{n-2}p \\
&= \sum_{n=1}^{\infty} (n+1)(1-p)^{n-1}p \\
&= 1 + \sum_{n=1}^{\infty} n(1-p)^{n-1}p \\
&= 1 + \mathrm{E}\left[X\right].
\end{aligned}$$

Since we know that $\mathrm{E}[X] = p + (1 + \mathrm{E}[X])(1-p)$, a bit of algebra gives $\mathrm{E}[X] = 1/p$, which is about what we'd expect.

There are more direct ways to get the same result. If we don't have conditional expectation to work with, we can try computing the sum $\mathrm{E}[X] = \sum_{n=1}^{\infty} n(1-p)^{n-1}p$ directly. The easiest way to do this is probably to use generating functions (see, for example, [GKP88, Chapter 7] or [Wil06]). An alternative argument is given in [MU05, §2.4]; this uses the fact that $\mathrm{E}[X] = \sum_{n=1}^{\infty} \Pr[X \geq n]$, which holds when X takes on only non-negative integer values.

3.6.2 Coupon collector

In the **coupon collector problem**, we throw balls uniformly and independently into n bins until every bin has at least one ball. When this happens, how many balls have we used on average?[8]

Let X_i be the number of balls needed to go from $i-1$ nonempty bins to i nonempty bins. It's easy to see that $X_1 = 1$ always. For larger i, each time we throw a ball, it lands in an empty bin with probability $\frac{n-i+1}{n}$. This

[8] The name comes from the problem of collecting coupons at random until you have all of them. A typical algorithmic application is having a cluster of machines choose jobs to finish at random from some list until all are done. The expected number of job executions to complete n jobs is given exactly by the solution to the coupon collector problem.

means that X_i has a geometric distribution with probability $\frac{n-i+1}{n}$, giving $\mathrm{E}[X_i] = \frac{n}{n-i+1}$ from the analysis in §3.6.1.

To get the total expected number of balls, take the sum

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=1}^{n} X_i\right] &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} \frac{n}{n-i+1} \\
&= n \sum_{i=1}^{n} \frac{1}{i} \\
&= nH_n.
\end{aligned}$$

In asymptotic terms, this is $\Theta(n \log n)$.

3.6.3 Hoare's FIND

Hoare's FIND [Hoa61b], often called **QuickSelect**, is an algorithm for finding the k-th smallest element of an unsorted array that works like QuickSort, only after partitioning the array around a random pivot we throw away the part that doesn't contain our target and recurse only on the surviving piece. As with QuickSort, we'd like to compute the expected number of comparisons used by this algorithm, on the assumption that the cost of the comparisons dominates the rest of the costs of the algorithm.

Here the indicator-variable trick gets painful fast. It turns out to be easier to get an upper bound by computing the expected number of elements that are left after each split.

First, let's analyze the pivot step. If the pivot is chosen uniformly, the number of elements X smaller than the pivot is uniformly distributed in the range 0 to $n-1$. The number of elements larger than the pivot will be $n - X - 1$. In the worst case, we find ourselves recursing on the large pile always, giving a bound on the number of survivors Y of $Y \leq \max(X, n-X+1)$.

What is the expected value of Y? By considering both ways the max can go, we get $\mathrm{E}[Y] = \mathrm{E}[X \mid X > n-X+1] \Pr[X > n-X+1] + \mathrm{E}[n-X+1 \mid n-X+1 \geq X]$. For both conditional expectations we are choosing a value uniformly in either the range $\left\lceil \frac{n-1}{2} \right\rceil$ to $n-1$ or $\left\lceil \frac{n-1}{2} \right\rceil + 1$ to $n-1$, and in either case the expectation will be equal to the average of the two endpoints by symmetry.

So we get

$$\begin{aligned}
\mathrm{E}[Y] &\leq \frac{n/2+n-1}{2}\Pr[X > n-X+1] + \frac{n/2+n}{2}\Pr[n-X+1 \geq X] \\
&= \left(\frac{3}{4}n - \frac{1}{2}\right)\Pr[X > n-X+1] + \frac{3}{4}n\Pr[n-X+1 \geq X] \\
&\leq \frac{3}{4}n.
\end{aligned}$$

Now let X_i be the number of survivors after i pivot steps. Note that $\max(0, X_i - 1)$ gives the number of comparisons at the following pivot step, so that $\sum_{i=0}^{\infty} X_i$ is an upper bound on the number of comparisons.

We have $X_0 = n$, and from the preceding argument $\mathrm{E}[X_1] \leq (3/4)n$. But more generally, we can use the same argument to show that $\mathrm{E}[X_{i+1} \mid X_i] \leq (3/4)X_i$, and by induction $\mathrm{E}[X_i] \leq (3/4)^i n$. We also have that $X_j = 0$ for all $j \geq n$, because we lose at least one element (the pivot) at each pivoting step. This saves us from having to deal with an infinite sum.

Using linearity of expectation,

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=0}^{\infty} X_i\right] &= \mathrm{E}\left[\sum_{i=0}^{n} X_i\right] \\
&= \sum_{i=0}^{n} \mathrm{E}[X_i] \\
&\leq \sum_{i=0}^{n} (3/4)^i n \\
&\leq 4n.
\end{aligned}$$

Chapter 4

Basic probabilistic inequalities

Here we're going to look at some inequalities useful for proving properties of randomized algorithms. These come in two flavors: inequalities involving probabilities, which are useful for bounding the probability that something bad happens, and inequalities involving expectations, which are used to bound expected running times. Later, in Chapter 5, we'll be doing both, by looking at inequalities that show that a random variable is close to its expectation with high probability.

4.1 Union bound (Boole's inequality)

For any countable collection of events $\{A_i\}$,

$$\Pr\left[\bigcup A_i\right] \leq \sum \Pr\left[A_i\right]. \tag{4.1.1}$$

The direct way to prove this is to replace A_i with $B_i = A_i \setminus \bigcup_{j=1}^{i-1} A_i$. Then $\bigcup A_i = \bigcup B_i$, but since the B_i are disjoint and each B_i is a subset of the corresponding A_i, we get $\Pr\left[\bigcup A_i\right] = \Pr\left[\bigcup B_i\right] = \sum \Pr\left[B_i\right] \leq \sum \Pr\left[A_i\right]$.

The typical use of the union bound is to show that if an algorithm can fail only if various improbable events occur, then the probability of failure is no greater than the sum of the probabilities of these events. This reduces the problem of showing that an algorithm works with probability $1 - \epsilon$ to constructing an **error budget** that divides the ϵ probability of failure among all the bad outcomes.

4.1.1 Applications

(See also the proof of Adleman's Theorem in Chapter 12.)

4.1.1.1 Balls in bins

Suppose we toss n balls into n bins. What is the likely value of the maximum number of balls in any one bin?[1]

For a single bin, the probability that it has k balls is $\binom{n}{k} n^{-k}(1-1/n)^{n-k} \leq \frac{n^k}{k!} n^{-k} = 1/k!$. The probability that it has more than k balls is bounded by $\sum_{i=k}^{\infty} \frac{1}{i!} \leq \sum_{i=k}^{\infty} \frac{1}{k!(k+1)^{i-k}} = \frac{1}{k!(1-1/(k+1))} \leq \frac{2}{k!}$.

Applying the union bound, we get that the probability that there is any bin with at least k balls is at most $2n/k!$. This suggests choosing k so that $k! \gg 2n$. Stirling's formula says that $k! \geq \sqrt{2\pi k}(k/e)^k \geq (k/e)^k$ or $\ln(k!) \geq k(\ln k - 1)$. If we set $k = c \ln n / \ln \ln n$, we get

$$\begin{aligned} \ln(k!) &\geq \frac{c \ln n}{\ln \ln n} (\ln c + \ln \ln n - \ln \ln \ln n - 1) \\ &\geq \frac{c \ln n}{2} \end{aligned}$$

when n is sufficiently large.

It follows that the bound $2n/k!$ in this case is less than $2n/\exp(c \ln n/2) = 2n \cdot n^{-c/2} = 2n^{1-c/2}$. For suitable choice of c we get a high probability that every bin gets at most $O(\log n / \log \log n)$ balls.

4.1.1.2 Independent sets

Given a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$, mark a subset of $n/2\sqrt{m}$ vertices uniformly at random. The probability that any particular vertex is marked is then $(n/2/\sqrt{m})/n = 1/2\sqrt{m}$, and the probability that both endpoints of an edge are marked is $(1/2\sqrt{m}) \cdot ((n/2/\sqrt{m} - 1)/n < 1/(2\sqrt{m}))^2 = 1/4m$. So the probability that at least one of the edges has two marked endpoints is at most $m/4m = 1/4$. We thus have a randomized algorithm that outputs a set of size $\frac{n}{2\sqrt{m}}$ on average that is an independent set with probability $3/4$, without looking at any of the edges.

To see how this compares with more sensible algorithms, note that for a graph with maximum degree Δ, we can construct and independent set of size $\frac{n}{\Delta+1}$ deterministically by marking some vertex, deleting all of its

[1] Algorithmic version: we insert n elements into a hash table with n positions using a random hash function. What is the maximum number of elements in any one position?

neighbors, and repeating until we run out of vertices. The randomized algorithm produces an independent set of size $\frac{n}{2\sqrt{nd}} = \sqrt{\frac{n}{2\Delta}}$ in this case, which is much worse.

4.2 Markov's inequality

This is the key tool for turning expectations of non-negative random variables into (upper) bounds on probabilities. Used directly, it generally doesn't give very good bounds, but it can work well if we apply it to $\mathrm{E}[f(X)]$ for a fast-growing function f; see Chebyshev's inequality (5.1.1) or Chernoff bounds (§5.2).

If $X \geq 0$, then

$$\Pr[X \geq \alpha] \leq \frac{\mathrm{E}[X]}{\alpha}. \tag{4.2.1}$$

The proof is immediate from the law of total probability (2.3.3). We have

$$\begin{aligned}\mathrm{E}[X] &= \mathrm{E}[X \mid X \geq \alpha]\Pr[X \geq \alpha] + \mathrm{E}[X \mid x < \alpha]\Pr[X < \alpha] \\ &\geq \alpha \Pr[X \geq \alpha];\end{aligned}$$

now solve for $\Pr[X \geq \alpha]$.

Markov's inequality doesn't work in reverse. For example, consider the following game: for each integer $k > 0$, with probability 2^{-k}, I give you 2^k dollars. Letting X be your payoff from the game, we have $\Pr\left[X \geq 2^k\right] = \sum_{j=k}^{\infty} 2^{-k} = 2^{-k+1} = \frac{2}{2^k}$. The right-hand side here is exactly what we would get from Markov's inequality if $\mathrm{E}[X] = 2$. But in this case, $\mathrm{E}[X] \neq 2$; in fact, the expectation of X is given by $\sum_{k=1}^{\infty} 2^k 2^{-k}$, which diverges.

4.2.1 Applications

4.2.1.1 The union bound

Combining Markov's inequality with linearity of expectation and indicator variables gives a succinct proof of the union bound:

$$\begin{aligned}\Pr\left[\bigcup A_i\right] &= \Pr\left[\sum 1_{A_i} \geq 1\right] \\ &\leq \mathrm{E}\left[\sum 1_{A_i}\right] \\ &= \sum \mathrm{E}[1_{A_i}] \\ &= \sum \Pr[A_i].\end{aligned}$$

Note that for this to work for infinitely many events we need to use the fact that 1_{A_i} is always non-negative.

4.2.1.2 Fair coins

Flip n independent fair coins, and let S be the number of heads we get. Since $\mathrm{E}[S] = n/2$, we get $\Pr[S = n] = 1/2$. This is much larger than the actual value 2^{-n}, but it's the best we can hope for if we only know $\mathrm{E}[S]$: if we let S be 0 or n with equal probability, we also get $\mathrm{E}[S] = n/2$.

4.2.1.3 Randomized QuickSort

The expected running time for randomized QuickSort is $O(n \log n)$. It follows that the probability that randomized QuickSort takes more than $f(n)$ time is $O(n \log n/f(n))$. For example, the probability that it performs the maximum $\binom{n}{2} = O(n^2)$ comparisons is $O(\log n/n)$. (It's possible to do much better than this.)

4.2.1.4 Balls in bins

Suppose we toss m balls in n bins, uniformly and independently. What is the probability that some particular bin contains at least k balls? The probability that a particular ball lands in a particular bin is $1/n$, so the expected number of balls in the bin is m/n. This gives a bound of m/nk that a bin contains k or more balls. We can combine this with the union bound to show that the probability that any bin contains k or more balls is at most m/k. Unfortunately this is not a very good bound.

4.3 Jensen's inequality

This is mostly useful if we can calculate $\mathrm{E}[X]$ easily for some X, but what we really care about is some other random variable $Y = f(X)$.

Jensen's inequality applies when f is a **convex function**, which means that for any x, y, and $0 \leq \mu \leq 1$, $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$. Geometrically, this means that the line segment between any two points on the graph of f never goes below f; i.e., that the set of points $\{(x, y) \mid y \geq f(x)\}$ is convex. If we want to show that f is convex, it's enough to show that that $f\left(\frac{x+y}{2}\right) \leq \frac{f(x)+f(y)}{2}$ for all x and y (in effect, we only need to prove it for the case $\lambda = 1/2$). If f is twice-differentiable, an even easier way is to show that $f''(x) \geq 0$ for all x.

The inequality says that if X is a random variable and f is convex then

$$f(\mathrm{E}[X]) \leq \mathrm{E}[f(X)]. \tag{4.3.1}$$

Alternatively, if f is **concave** (which means that $f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$, or equivalently that $-f$ is convex), the reverse inequality holds:

$$f(\mathrm{E}[X]) \geq \mathrm{E}[f(X)]. \tag{4.3.2}$$

The intuition in both cases is that the basic definition of convexity or concavity is just Jensen's inequality for the random variable X that equals x with probability λ and y with probability $1-\lambda$, but that the more general result holds for any random variable for which the expectations exist.

4.3.1 Applications

4.3.1.1 Fair coins: lower bound

Suppose we flip n fair coins, and we want to get a lower bound on $\mathrm{E}\left[X^2\right]$, where X is the number of heads. The function $f : x \mapsto x^2$ is convex (take its second derivative), so (4.3.1) gives $\mathrm{E}\left[X^2\right] \geq (\mathrm{E}[X])^2 = \frac{n^2}{4}$. (The actual value for $\mathrm{E}\left[X^2\right]$ is $\frac{n^2}{4} + \frac{n}{4}$, which can be found using generating functions.[2])

4.3.1.2 Fair coins: upper bound

For an upper bound, we can choose a concave f. For example, if X is as in the previous example, $\mathrm{E}[\lg X] \leq \lg \mathrm{E}[X] = \lg \frac{n}{2} = \lg n - 1$. This is probably pretty close to the exact value, as we will see later that X will almost always be within a factor of $1 + o(1)$ of $n/2$. It's not a terribly useful upper bound, because if we use it with (say) Markov's inequality, the most we can prove is that $\Pr[X = n] = \Pr[\lg X = \lg n] \leq \frac{\lg n - 1}{\lg n} = 1 - \frac{1}{\lg n}$, which is an even worse bound than the $1/2$ we can get from applying Markov's inequality to X directly.

[2] Here's how: The **probability generating function** for X is $F(z) = \mathrm{E}\left[z^k\right] = \sum z^k \Pr[X = k] = 2^{-n}(1+z)^n$. Then $zF'(z) = 2^{-n}nz(1+z)^{n-1} = \sum_k kz^k \Pr[X = k]$. Taking the derivative of this a second time gives $2^{-n}n(1+z)^{n-1} + 2^{-n}n(n-1)z(1+z)^{n-2} = \sum_k k^2 z^{k-1} \Pr[X = k]$. Evaluate this monstrosity at $z = 1$ to get $\mathrm{E}\left[X^2\right] = \sum_k k^2 \Pr[X = k] = 2^{-n}n2^{n-1} + 2^{-n}n(n-1)2^{n-2} = n/2 + n(n-1)/4 = \frac{2n+n^2-n}{4} = n^2/4 + n/4$. This is pretty close to the lower bound we got out of Jensen's inequality, but we can't count on this happening in general.

4.3.1.3 Sifters

Here's an example of Jensen's inequality in action in the analysis of an actual distributed algorithm. For some problems in distributed computing, it's useful to reduce coordinating a large number of processes to coordinating a smaller number. A **sifter** [AA11] is a randomized mechanism for an asynchronous shared-memory system that sorts the processes into "winners" and "losers," guaranteeing that there is at least one winner. The goal is to make the expected number of winners as small as possible. The problem is tricky, because processes can only communicate by reading and writing shared variables, and an adversary gets to choose which processes participate and fix the schedule of when each of these processes perform their operations.

The current best known sifter is due to Giakkoupis and Woelfel [GW12]. For n processes, it uses an array A of $\lceil \lg n \rceil$ bits, each of which can be read or written by any of the processes. When a process executes the sifter, it chooses a random index $r \in 1 \ldots \lceil \lg n \rceil$ with probability 2^{-r-1} (this doesn't exactly sum to 1, so the excess probability gets added to $r = \lceil \lg n \rceil$). The process then writes a 1 to $A[r]$ and reads $A[r+1]$. If it sees a 0 in its read (or chooses $r = \lceil \lg n \rceil$), it wins; otherwise it loses.

This works as a sifter, because no matter how many processes participate, some process chooses a value of r at least as large as any other process's value, and this process wins. To bound the expected number of winners, take the sum over all r over the random variable W_r representing the winners who chose this particular value r. A process that chooses r wins if it carries out its read operation before any process writes $r+1$. If the adversary wants to maximize the number of winners, it should let each process read as soon as possible; this effectively means that a process that choose r wins if no process previously chooses $r+1$. Since r is twice as likely to be chosen as $r+1$, conditioning on a process picking r or $r+1$, there is only a $1/3$ chance that it chooses $r+1$. So at most $1/(1/3) - 1 = 2 = O(1)$ process on average choose r before some process chooses $r+1$. (A simpler argument shows that the expected number of processes that win because they choose $r = \lceil \lg n \rceil$ is at most 2 as well.)

Summing $W_r \leq 2$ over all r gives at most $2 \lceil \lg n \rceil$ winners on average. Furthermore, if $k < n$ processes participate, essentially the same analysis shows that only $2 \lceil \lg k \rceil$ processes win on average. So this is a pretty effective tool for getting rid of excess processes.

But it gets better. Suppose that we take the winners of one sifter and feed them into a second sifter. Let X_k be the number of processes left after k sifters. We have that $X_0 = n$ and $\mathrm{E}[X_1] \leq 2 \lceil \lg n \rceil$, but what can we

say about $\mathrm{E}[X_2]$? We can calculate $\mathrm{E}[X_2] = \mathrm{E}[\mathrm{E}[X_2 \mid X_1]] \leq 2\lceil \lg X_1 \rceil$. Unfortunately, the ceiling means that $2\lceil \lg x \rceil$ is not a concave function, but $f(x) = 2(\lg x + 1) \geq 2\lceil \lg x \rceil$ is. So $\mathrm{E}[X_2] \leq f(f(n))$, and in general $\mathrm{E}[X_i] \leq f^{(i)}(n)$, where $f^{(i)}$ is the i-fold composition of f. All the extra constants obscure what is going on a bit, but with a little bit of algebra it is not too hard to show that $f^{(i)}(n) = O(1)$ for $i = O(\log^* n)$.[3] So this gets rid of all but a constant number of processes very quickly.

[3] The $\log^*$ function counts how many times you need to hit n with lg to reduce it to one or less. So $\log^* 1 = 0, \log^* 2 = 1, \log^* 4 = 2, \log^* 16 = 3, \log^* 65536 = 4, \log^* 2^{65536} = 5$, and after that it starts getting silly.

Chapter 5

Concentration bounds

If we really want to get tight bounds on a random variable X, the trick will turn out to be picking some non-negative function $f(X)$ where (a) we can calculate $\mathrm{E}[f(X)]$, and (b) f grows fast enough that merely large values of X produce huge values of $f(X)$, allowing us to get small probability bounds by applying Markov's inequality to $f(X)$. This approach is often used to show that X lies close to $\mathrm{E}[X]$ with reasonably high probability, what is known as a **concentration bound**.

Typically concentration bounds are applied to sums of random variables, which may or may not be fully independent. Which bound you may want to use often depends on the structure of your sum. A quick summary of the bounds in this chapter is given in Table 5.1. The rule of thumb is to use Chernoff bounds (§5.2) if you have a sum of independent 0–1 random variables, the Azuma-Hoeffding inequality (§5.3) if you have bounded variables with a more complicated distribution that may be less independent, and Chebyshev's inequality (§5.1) if nothing else works but you can somehow compute the variance of your sum (e.g., if the X_i are independent or have easily computed covariance). In the case of Chernoff bounds, you will almost always end up using one of the weaker but cleaner versions in §5.2.2 rather than the general version in §5.2.1.

5.1 Chebyshev's inequality

Chebyshev's inequality allows us to show that a random variable is close to its mean, by applying Markov's inequality to the **variance** of X, given by $\mathrm{Var}[X] = \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] = \mathrm{E}\left[X^2 - 2X\,\mathrm{E}[X] + (\mathrm{E}[X])^2\right] = \mathrm{E}\left[X^2\right] - (\mathrm{E}[X])^2$. It's a fairly weak concentration bound, that is most useful when

Chernoff	$X_i \in \{0,1\}$, independent	$\Pr[S \geq (1+\delta)\operatorname{E}[S]] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\operatorname{E}[S]}$
Azuma-Hoeffding	$\lvert X_i\rvert \leq c_i$, martingale	$\Pr[S \geq t] \leq \exp\left(-\frac{t^2}{2\sum c_i^2}\right)$
Chebyshev		$\Pr[\lvert S - \operatorname{E}[S]\rvert \geq \alpha] \leq \frac{\operatorname{Var}[S]}{\alpha^2}$

Table 5.1: Concentration bounds for $S = \sum X_i$ (strongest to weakest)

X is a sum of random variables with limited independence.

Using Markov's inequality, calculate

$$\begin{aligned}\Pr[\lvert X - \operatorname{E}[X]\rvert \geq \alpha] &= \Pr\left[(X - \operatorname{E}[X])^2 \geq \alpha^2\right] \\ &\leq \frac{\operatorname{E}\left[(X - \operatorname{E}[X])^2\right]}{\alpha^2} \\ &= \frac{\operatorname{Var}[X]}{\alpha^2}. \end{aligned} \tag{5.1.1}$$

5.1.1 Computing variance

At this point it would be reasonable to ask why we are going through $\operatorname{Var}[X] = \operatorname{E}\left[(X - \operatorname{E}[X])^2\right]$ rather than just using $\operatorname{E}[\lvert X - \operatorname{E}[X]\rvert]$. The reason is that $\operatorname{Var}[X]$ is usually easier to compute, especially if X is a sum. In this section, we'll give some examples of computing variance, including for various standard random variables that come up a lot in randomized algorithms.

5.1.1.1 Alternative formula

The first step is to give an alternative formula for the variance that is more convenient in some cases.

Expand

$$\begin{aligned}\operatorname{E}\left[(X - \operatorname{E}[X])^2\right] &= \operatorname{E}\left[X^2 - 2X \cdot \operatorname{E}[X] + (\operatorname{E}[X])^2\right] \\ &= \operatorname{E}\left[X^2\right] - 2\operatorname{E}[X] \cdot \operatorname{E}[X] + (\operatorname{E}[X])^2 \\ &= \operatorname{E}\left[X^2\right] - (\operatorname{E}[X])^2 .\end{aligned}$$

This formula is easier to use if you are estimating the variance from a sequence of samples; by tracking $\sum x_i^2$ and $\sum x_i$, you can estimate $\mathrm{E}\left[X^2\right]$ and $\mathrm{E}\left[X\right]$ in a single pass, without having to estimate $\mathrm{E}\left[X\right]$ first and then go back for a second pass to calculate $(x_i - \mathrm{E}\left[X\right])^2$ for each sample. We won't use this particular application much, but this explains why the formula is popular with statisticians.

5.1.1.2 Variance of a Bernoulli random variable

Recall that a Bernoulli random variable is 1 with probability p and 0 with probability $q = 1 - p$; in particular, any indicator variable is a Bernoulli random variable.

The variance of a Bernoulli random variable is easily calculated from the alternative formula:

$$\begin{aligned}\mathrm{Var}\left[X\right] &= \mathrm{E}\left[X^2\right] - \left(\mathrm{E}\left[X\right]\right)^2 \\ &= p - p^2 \\ &= pq.\end{aligned}$$

5.1.1.3 Variance of a sum

If $S = \sum_i X_i$, then we can calculate

$$\begin{aligned}\mathrm{Var}\left[S\right] &= \mathrm{E}\left[\left(\sum_i X_i\right)^2\right] - \left(\mathrm{E}\left[\sum_i X_i\right]\right)^2 \\ &= \mathrm{E}\left[\sum_i \sum_j X_i X_j\right] - \sum_i \sum_j \mathrm{E}\left[X_i\right]\mathrm{E}\left[X_j\right] \\ &= \sum_i \sum_j \left(\mathrm{E}\left[X_i X_j\right] - \mathrm{E}\left[X_i\right]\mathrm{E}\left[X_j\right]\right).\end{aligned}$$

For any two random variables X and Y, the quantity $\mathrm{E}\left[XY\right] - \mathrm{E}\left[X\right]\mathrm{E}\left[Y\right]$ is called the **covariance** of X and Y, written $\mathrm{Cov}\left[X, Y\right]$. We can use

$\operatorname{Cov}[X, Y]$ to rewrite the above expansion as

$$\operatorname{Var}\left[\sum_i X_i\right] = \sum_{i,j} \operatorname{Cov}[X_i, X_j] \tag{5.1.2}$$

$$= \sum_i \operatorname{Var}[X_i] + \sum_{i \neq j} \operatorname{Cov}[X_i, X_j] \tag{5.1.3}$$

$$= \sum_i \operatorname{Var}[X_i] + 2 \sum_{i<j} \operatorname{Cov}[X_i, X_j] \tag{5.1.4}$$

Note that $\operatorname{Cov}[X, Y] = 0$ when X and Y are independent; this makes Chebyshev's inequality particularly useful for **pairwise-independent** random variables, because then we can just sum up the variances of the individual variables.

A typical application is when we have a sum $S = \sum X_i$ of non-negative random variables with small covariance; here applying Chebyshev's inequality to S can often be used to show that S is not likely to be much smaller than $\operatorname{E}[S]$, which can be handy if we want to show that some lower bound holds on S with some probability. This complements Markov's inequality, which can only be used to get upper bounds.

More generally, the approach of bounding S from below by estimating $\operatorname{E}[S]$ and either $\operatorname{E}\left[S^2\right]$ or $\operatorname{Var}[S]$ is known as the **second-moment method**. In some cases, tighter bounds can be obtained by more careful analysis.

5.1.1.4 Variance of a geometric random variable

Let X be a geometric random variable with parameter p as defined in §3.6.1, so that X takes on the values $1, 2, \ldots$ and $\Pr[X = n] = q^{n-1}p$, where $q = 1 - p$ as usual. What is $\operatorname{Var}[X]$?

We know that $\operatorname{E}[X] = 1/p$, so $(\operatorname{E}[X])^2 = 1/p^2$. Computing $\operatorname{E}\left[X^2\right]$ is trickier. Rather than do this directly from the definition of expectation, we can exploit the memorylessness of geometric random variables to get it using conditional expectations, just like we did for $\operatorname{E}[X]$ in §3.6.1.

Conditioning on the two possible outcomes of the first trial, we have

$$\operatorname{E}\left[X^2\right] = p + q \operatorname{E}\left[X^2 \mid X > 1\right]. \tag{5.1.5}$$

We now argue that $\operatorname{E}\left[X^2 \mid X > 1\right] = \operatorname{E}\left[(X+1)^2\right]$. The intuition is that once we have flipped one coin the wrong way, we are back where we

started, except that now we have to add that extra coin to X. More formally, we have, for $n > 1$, $\Pr\left[X^2 = n \mid X > 1\right] = \frac{\Pr\left[X^2=n\right]}{\Pr[X>1]} = \frac{q^{n-1}p}{q} = q^{n-2}p = \Pr\left[X = n-1\right] = \Pr\left[X+1 = n\right]$. So we get the same probability mass function for X conditioned on $X > 1$ as for $X+1$ with no conditioning.

Applying this observation to the right-hand side of (5.1.5) gives

$$\begin{aligned}
\mathrm{E}\left[X^2\right] &= p + q\,\mathrm{E}\left[(X+1)^2\right] \\
&= p + q\left(\mathrm{E}\left[X^2\right] + 2\,\mathrm{E}\left[X\right] + 1\right) \\
&= p + q\,\mathrm{E}\left[X^2\right] + \frac{2q}{p} + q \\
&= 1 + q\,\mathrm{E}\left[X^2\right] + \frac{2q}{p}.
\end{aligned}$$

A bit of algebra turns this into

$$\begin{aligned}
\mathrm{E}\left[X^2\right] &= \frac{1+2q/p}{1-q} \\
&= \frac{1+2q/p}{p} \\
&= \frac{p+2q}{p^2} \\
&= \frac{2-p}{p^2}.
\end{aligned}$$

Now subtract $\left(\mathrm{E}\left[X\right]\right)^2 = \frac{1}{p^2}$ to get

$$\mathrm{Var}\left[X\right] = \frac{1-p}{p^2} = \frac{q}{p^2}. \tag{5.1.6}$$

By itself, this doesn't give very good bounds on X. For example, if we

want to bound the probability that $X = 1$, we get

$$\begin{aligned}
\Pr[X = 1] &= \Pr\left[X - \mathrm{E}[X] = 1 - \frac{1}{p}\right] \\
&\leq \Pr\left[|X - \mathrm{E}[X]| \geq \frac{1}{p} - 1\right] \\
&\leq \frac{\mathrm{Var}[X]}{\left(\frac{1}{p} - 1\right)^2} \\
&= \frac{q/p^2}{\left(\frac{1}{p} - 1\right)^2} \\
&= \frac{q}{(1-p)^2} \\
&= \frac{1}{q}.
\end{aligned}$$

Since $\frac{1}{q} \geq 1$, we could have gotten this bound with a lot less work.

The other direction is not much better. We can easily calculate that $\Pr[X \geq n]$ is exactly q^{n-1} (because this corresponds to flipping n coins the wrong way, no matter what happens with subsequent coins). Using Chebyshev's inequality gives

$$\begin{aligned}
\Pr[X \geq n] &\leq \Pr\left[\left|X - \frac{1}{p}\right| \geq n - \frac{1}{p}\right] \\
&\leq \frac{q/p^2}{\left(n - \frac{1}{p}\right)^2} \\
&= \frac{q}{(np-1)^2}.
\end{aligned}$$

This at least has the advantage of dropping below 1 when n gets large enough, but it's only polynomial in n while the true value is exponential.

Where this might be useful is in analyzing the sum of a bunch of geometric random variables, as occurs in the Coupon Collector problem discussed in §3.6.2.[1] Letting X_i be the number of balls to take us from $i-1$ to i empty bins, we have previously argued that X_i has a geometric distribution with

[1] We are following here a similar analysis in [MU05, §3.3.1].

$p = \frac{n-i-1}{n}$, so

$$\begin{aligned}\operatorname{Var}[X_i] &= \frac{i-1}{n} \Big/ \left(\frac{n-i-1}{n}\right)^2 \\ &= n\frac{i-1}{(n-i-1)^2},\end{aligned}$$

and

$$\begin{aligned}\operatorname{Var}\left[\sum_{i=1}^{n} X_i\right] &= \sum_{i=1}^{n} \operatorname{Var}[X_i] \\ &= \sum_{i=1}^{n} n\frac{i-1}{(n-i-1)^2}.\end{aligned}$$

Having the numerator go up while the denominator goes down makes this a rather unpleasant sum to try to solve directly. So we will follow the lead of Mitzenmacher and Upfal and bound the numerator by n, giving

$$\begin{aligned}\operatorname{Var}\left[\sum_{i=1}^{n} X_i\right] &\leq \sum_{i=1}^{n} n\frac{n}{(n-i-1)^2} \\ &= n^2 \sum_{i=1}^{n} \frac{1}{i^2} \\ &\leq n^2 \sum_{i=1}^{\infty} \frac{1}{i^2} \\ &= n^2 \frac{\pi^2}{6}.\end{aligned}$$

The fact that $\sum_{i=1}^{\infty} \frac{1}{i^2}$ converges to $\frac{\pi^2}{6}$ is not trivial to prove, and was first shown by Leonhard Euler in 1735 some ninety years after the question was first proposed.[2] But it's easy to show that the series converges to something, so even if we didn't have Euler's help, we'd know that the variance is $O(n^2)$.

Since the expected value of the sum is $\Theta(n \log n)$, this tells us that we are likely to see a total waiting time reasonably close to this; with at least constant probability, it will be within $\Theta(n)$ of the expectation. In fact, the distribution is much more sharply concentrated (see [MU05, §5.4.1] or [MR95, §3.6.3]), but this bound at least gives us something.

[2] See `http://en.wikipedia.org/wiki/Basel_Problem` for a history, or Euler's original paper [Eul68], available at `http://eulerarchive.maa.org/docs/originals/E352.pdf` for the actual proof in the full glory of its original 18th-century typesetting. Curiously, though Euler announced his result in 1735, he didn't submit the journal version until 1749, and it didn't see print until 1768. Things moved more slowly in those days.

5.1.2 More examples

Here are some more examples of Chebyshev's inequality in action. Some of these repeat examples for which we previously got crummier bounds in §4.2.1.

5.1.2.1 Flipping coins

Let X be the sum of n independent fair coins. Let X_i be the indicator variable for the event that the i-th coin comes up heads. Then $\operatorname{Var}[X_i] = 1/4$ and $\operatorname{Var}[X] = \sum \operatorname{Var}[X_i] = n/4$. Chebyshev's inequality gives $\Pr[X = n] \leq \Pr[|X - n/2| \geq n/2] \leq \frac{n/4}{(n/2)^2} = \frac{1}{n}$. This is still not very good, but it's getting better. It's also about the best we can do given only the assumption of pairwise independence.

To see this, let $n = 2^m - 1$ for some m, and let $Y_1 \ldots Y_m$ be independent, fair 0–1 random variables. For each non-empty subset S of $\{1 \ldots m\}$, let X_S be the exclusive OR of all Y_i for $i \in S$. Then (a) the X_i are pairwise independent; (b) each X_i has variance $1/4$; and thus (c) the same Chebyshev's inequality analysis for independent coin flips above applies to $X = \sum_S X_S$, giving $\Pr[|X - n/2| = n/2] \leq \frac{1}{n}$. In this case it is not actually possible for X to equal n, but we can have $X = 0$ if all the Y_i are 0, which occurs with probability $2^{-m} = \frac{1}{n+1}$. So the Chebyshev's inequality is almost tight in this case.

5.1.2.2 Balls in bins

Let X_i be the indicator that the i-th of m balls lands in a particular bin. Then $\operatorname{E}[X_i] = 1/n$, giving $\operatorname{E}[\sum X_i] = m/n$, and $\operatorname{Var}[X_i] = 1/n - 1/n^2$, giving $\operatorname{Var}[\sum X_i] = m/n - m/n^2$. So the probability that we get $k + m/n$ or more balls in a particular bin is at most $(m/n - m/n^2)/k^2 < m/nk^2$, and applying the union bound, the probability that we get $k + m/n$ or more balls in any of the n bins is less than m/k^2. Setting this equal to ϵ and solving for k gives a probability of at most ϵ of getting more than $m/n + \sqrt{m/\epsilon}$ balls in any of the bins. This is not as good a bound as we will be able to prove later, but it's at least non-trivial.

5.1.2.3 Lazy select

This is the example from [MR95, §3.3]; essentially the same example, specialized to finding the median, also appears in [MU05, §3.4].[3]

We want to find the k-th smallest element $S_{(k)}$ of a set S of size n. (The parentheses around the index indicate that we are considering the sorted version of the set $S_{(1)} < S_{(2)} \cdots < S_{(n)}$.) The idea is to:

1. Sample a multiset R of $n^{3/4}$ elements of S with replacement and sort them. This takes $O(n^{3/4} \log n^{3/4}) = o(n)$ comparisons so far.

2. Use our sample to find an interval that is likely to contain $S_{(k)}$. The idea is to pick indices $\ell = (k - n^{3/4})n^{-1/4}$ and $r = (k + n^{3/4})n^{-1/4}$ and use $R_{(\ell)}$ and $R_{(r)}$ as endpoints (we are omitting some floors and maxes here to simplify the notation; for a more rigorous presentation see [MR95]). The hope is that the interval $P = [R_{(\ell)}, R(r)]$ in S will both contain $S_{(k)}$, and be small, with $|P| \leq 4n^{3/4} + 2$. We can compute the elements of P in $2n$ comparisons exactly by comparing every element with both $R_{(\ell)}$ and $R_{(r)}$.

3. If both these conditions hold, sort P ($o(n)$ comparisons) and return $S_{(k)}$. If not, try again.

We want to get a bound on how likely it is that P either misses $S_{(k)}$ or is too big.

For any fixed k, the probability that one sample in R is less than or equal to $S_{(k)}$ is exactly k/n, so the expected number X of samples $\leq S_{(k)}$ is exactly $kn^{-1/4}$. The variance on X can be computed by summing the variances of the indicator variables that each sample is $\leq S_{(k)}$, which gives a bound $\operatorname{Var}[X] = n^{3/4}((k/n)(1 - k/n)) \leq n^{3/4}/4$. Applying Chebyshev's inequality gives $\Pr\left[\left|X - kn^{-1/4}\right| \geq \sqrt{n}\right] \leq n^{3/4}/4n = n^{-1/4}/4$.

Now let's look at the probability that P misses $S_{(k)}$ because $R_{(\ell)}$ is too big, where $\ell = kn^{-1/4} - \sqrt{n}$. This is

$$\begin{aligned}\Pr\left[R_{(\ell)} > S_{(k)}\right] &= \Pr\left[X < kn^{-1/4} - \sqrt{n}\right] \\ &\leq n^{-1/4}/4.\end{aligned}$$

[3] The history of this example is that Motwani and Raghavan adapted this algorithm from a similar algorithm by Floyd and Rivest [FR75]. Mitzenmacher and Upfal give a version that also includes the adaptations appearing Motwani and Raghavan, although they don't say where they got it from, and it may be that both textbook versions come from a common folklore source.

(with the caveat that we are being sloppy about round-off errors).

Similarly,

$$\begin{aligned}\Pr\left[R_{(h)} < S_{(k)}\right] &= \Pr\left[X > kn^{-1/4} + \sqrt{n}\right] \\ &\leq n^{-1/4}/4.\end{aligned}$$

So the total probability that P misses $S_{(k)}$ is at most $n^{-1/4}/2$.

Now we want to show that $|P|$ is small. We will do so by showing that it is likely that $R_{(\ell)} \geq S_{(k-2n^{3/4})}$ and $R_{(h)} \leq S_{(k+2n^{3/4})}$. Let X_ℓ be the number of samples in R that are $\leq S_{(k-2n^{3/4})}$ and X_r be the number of samples in R that are $\leq S_{(k+2n^{3/4})}$. Then we have $\mathrm{E}[X_\ell] = kn^{-1/4} - 2\sqrt{n}$ and $\mathrm{E}[X_r] = kn^{-1/4} + 2\sqrt{n}$, and $\mathrm{Var}[X_l]$ and $\mathrm{Var}[X_r]$ are both bounded by $n^{3/4}/4$.

We can now compute

$$\Pr\left[R_{(l)} < S_{(k-2n^{3/4})}\right] = Pr[X_l > kn^{-1/4} - \sqrt{n}] < n^{-1/4}/4$$

by the same Chebyshev's inequality argument as before, and get the symmetric bound on the other side for $\Pr\left[R_{(r)} > S_{(k+2n^{3/4})}\right]$. This gives a total bound of $n^{-1/4}/2$ that P is too big, for a bound of $n^{-1/4} = o(n)$ that the algorithm fails on its first attempt.

The total expected number of comparisons is thus given by $T(n) = 2n + o(n) + O(n^{-1/4}T(n)) = 2n + o(n)$.

5.2 Chernoff bounds

To get really tight bounds, we apply Markov's inequality to $\exp(\alpha S)$, where $S = \sum_i X_i$. This works best when the X_i are independent: if this is the case, so are the variables $\exp(\alpha X_i)$, and so we can easily calculate $\mathrm{E}[\exp(\alpha S)] = \mathrm{E}\left[\prod_i \exp(\alpha X_i)\right] = \prod_i \mathrm{E}[\exp(\alpha X_i)]$.

The quantity $\mathrm{E}[\exp(\alpha S)]$, treated as a function of α, is called the **moment generating function** of S, because it expands formally into $\sum_{k=0}^{\infty} \mathrm{E}\left[X^k\right] \frac{\alpha^k}{k!}$, the **exponential generating function** for the series of k-th **moments** $\mathrm{E}\left[X^k\right]$. Note that it may not converge for all S and α;[4] we will be careful to choose α for which it does converge and for which Markov's inequality gives us good bounds.

[4]For example, the moment generating function for our earlier bad X with $\Pr\left[X = 2^k\right] = 2^{-k}$ is equal to $\sum_k 2^{-k} e^{\alpha k}$, which diverges unless $e^\alpha/2 < 1$.

5.2.1 The classic Chernoff bound

The basic Chernoff bound applies to sums of independent 0–1 random variables, which need not be identically distributed. For identically distributed random variables, the sum has a binomial distribution, which we can either compute exactly or bound more tightly using approximations specific to binomial tails; for sums of bounded random variables that aren't necessarily 0–1, we can use Hoeffding's inequality instead (see §5.3).

Let each X_i for $i = 1 \ldots n$ be a 0–1 random variable with expectation p_i, so that $\mathrm{E}[S] = \mu = \sum_i p_i$. Recall that the plan is to show $\Pr[S \geq (1+\delta)\mu]$ is small when δ and μ are large, by applying Markov's inequality to $\mathrm{E}\left[e^{\alpha S}\right]$, where α will be chosen to make the bound as tight as possible for given δ. The first step is to get an upper bound on $\mathrm{E}\left[e^{\alpha S}\right]$.

Compute

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha S}\right] &= \mathrm{E}\left[e^{\alpha \sum X_i}\right] \\
&= \prod_i \mathrm{E}\left[e^{\alpha X_i}\right] \\
&= \prod_i \left(p_i e^{\alpha} + (1-p_i)e^0\right) \\
&= \prod_i \left(p_i e^{\alpha} + 1 - p_i\right) \\
&= \prod_i \left(1 + \left(e^{\alpha} - 1\right) p_i\right) \\
&\leq \prod_i e^{(e^{\alpha}-1)p_i} \\
&= e^{(e^{\alpha}-1)\sum_i p_i} \\
&= e^{(e^{\alpha}-1)\mu}.
\end{aligned}$$

The sneaky inequality step in the middle uses the fact that $(1+x) \leq e^x$ for all x, which itself is one of the most useful inequalities you can memorize.[5]

What's nice about this derivation is that at the end, the p_i have vanished. We don't care what random variables we started with or how many of them there were, but only about their expected sum μ.

Now that we have an upper bound on $\mathrm{E}\left[e^{\alpha S}\right]$, we can throw it into

[5] For a proof of this inequality, observe that the function $f(x) = e^x - (1+x)$ has the derivative $e^x - 1$, which is positive for $x > 0$ and negative for $x < 0$. It follows that $x = 1$ is the unique minimum of f, at which $f(1) = 0$.

Markov's inequality to get the bound we really want:

$$\begin{aligned}
\Pr\left[S \geq (1+\delta)\mu\right] &= \Pr\left[e^{\alpha S} \geq e^{\alpha(1+\delta)\mu}\right] \\
&\leq \frac{\mathrm{E}\left[e^{\alpha S}\right]}{e^{\alpha(1+\delta)\mu}} \\
&\leq \frac{e^{(e^{\alpha}-1)\mu}}{e^{\alpha(1+\delta)\mu}} \\
&= \left(\frac{e^{e^{\alpha}-1}}{e^{\alpha(1+\delta)}}\right)^{\mu}. \\
&= \left(e^{e^{\alpha}-1-\alpha(1+\delta)}\right)^{\mu}.
\end{aligned}$$

We now choose α to minimize the base in the last expression, by minimizing its exponent $e^{\alpha}-1-\alpha(1+\delta)$. Setting the derivative of this expression with respect to α to zero gives $e^{\alpha} = (1+\delta)$ or $\alpha = \ln(1+\delta)$; luckily, this value of α is indeed greater than 0 as we have been assuming. Plugging this value in gives

$$\begin{aligned}
\Pr\left[S \geq (1+\delta)\mu\right] &\leq \left(e^{(1+\delta)-1-(1+\delta)\ln(1+\delta)}\right)^{\mu} \\
&= \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}. \qquad (5.2.1)
\end{aligned}$$

The base of this rather atrocious quantity is $e^0/1^1 = 1$ at $\delta = 0$, and its derivative is negative for $\delta \geq 0$ (the easiest way to show this is to substitute $\delta = x-1$ first). So the bound is never greater than 1 and is both decreasing and less than 1 as soon as $\delta > 0$. We also have that the bound is exponential in μ for any fixed δ.

If we look at the shape of the base as a function of δ, we can observe that when δ is very large, we can replace $(1+\delta)^{1+\delta}$ with δ^{δ} without changing the bound much (and to the extent that we change it, it's an increase, so it still works as a bound). This turns the base into $\frac{e^{\delta}}{\delta^{\delta}} = (e/\delta)^{\delta} = 1/(\delta/e)^{\delta}$. This is pretty close to Stirling's formula for $1/\delta!$ (there is a $\sqrt{2\pi\delta}$ factor missing).

For very small δ, we have that $1+\delta \approx e^{\delta}$, so the base becomes approximately $\frac{e^{\delta}}{e^{\delta(1+\delta)}} = e^{-\delta^2}$. This approximation goes in the wrong direction (it's smaller than the actual value) but with some fudging we can show bounds of the form $e^{-\mu\delta^2/c}$ for various constants c, as long as δ is not too big.

5.2.2 Easier variants

The full Chernoff bound can be difficult to work with, especially since it's hard to invert (5.2.1) to find a good δ that gives a particular ϵ bound. Fortunately, there are approximate variants that substitute a weaker but less intimidating bound. Some of the more useful are:

- For $0 \leq \delta \leq 1.81$,

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/3}. \tag{5.2.2}$$

  (The actual upper limit is slightly higher.) Useful for small values of δ, especially because the bound can be inverted: if we want $\Pr[X \geq (1+\delta)\mu] \leq \exp(-\mu\delta^2/3) \leq \epsilon$, we can use any δ with $\sqrt{3\ln(1/\epsilon)/\mu} \leq \delta \leq 1.81$. The essential idea to the proof is to show that, in the given range, $e^\delta/(1+\delta)^{1+\delta} \leq \exp(-\delta^2/3)$. This is easiest to do numerically; a somewhat more formal argument that the bound holds in the range $0 \leq \delta \leq 1$ can be found in [MU05, Theorem 4.4].

- For $0 \leq \delta \leq 4.11$,

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\mu\delta^2/4}. \tag{5.2.3}$$

  This is a slightly weaker bound than the previous that holds over a larger range. It gives $\Pr[X \geq (1+\delta)\mu] \leq \epsilon$ if $\sqrt{4\ln(1/\epsilon)/\mu} \leq \delta \leq 4.11$. Note that the version given on page 72 of [MR95] is *not correct*; it claims that the bound holds up to $\delta = 2e - 1 \approx 4.44$, but it fails somewhat short of this value.

- For $R \geq 2e\mu$,

$$\Pr[X \geq R] \leq 2^{-R}. \tag{5.2.4}$$

  Sometimes the assumption is replaced with the stronger $R \geq 6\mu$ (this is the version given in [MU05, Theorem 4.4], for example); one can also verify numerically that $R \geq 5\mu$ (i.e., $\delta \geq 4$) is enough. The proof of the $2e\mu$ bound is that $e^\delta/(1+\delta)^{(1+\delta)} < e^{1+\delta}/(1+\delta)^{(1+\delta)} = (e/(1+\delta))^{1+\delta} \leq 2^{-(1+\delta)}$ when $e/(1+\delta) \leq 1/2$ or $\delta \geq 2e-1$. Raising this to μ gives $\Pr[X \geq (1+\delta)\mu] \leq 2^{-(1+\delta)\mu}$ for $\delta \geq 2e-1$. Now substitute R for $(1+\delta)\mu$ (giving $R \geq 2e\mu$) to get the full result. Inverting this one gives $\Pr[X \geq R] \leq \epsilon$ when $R \geq \min(2e\mu, \lg(1/\epsilon))$.

Figure 5.1 shows the relation between the various bounds when $\mu = 1$, in the region where they cross each other.

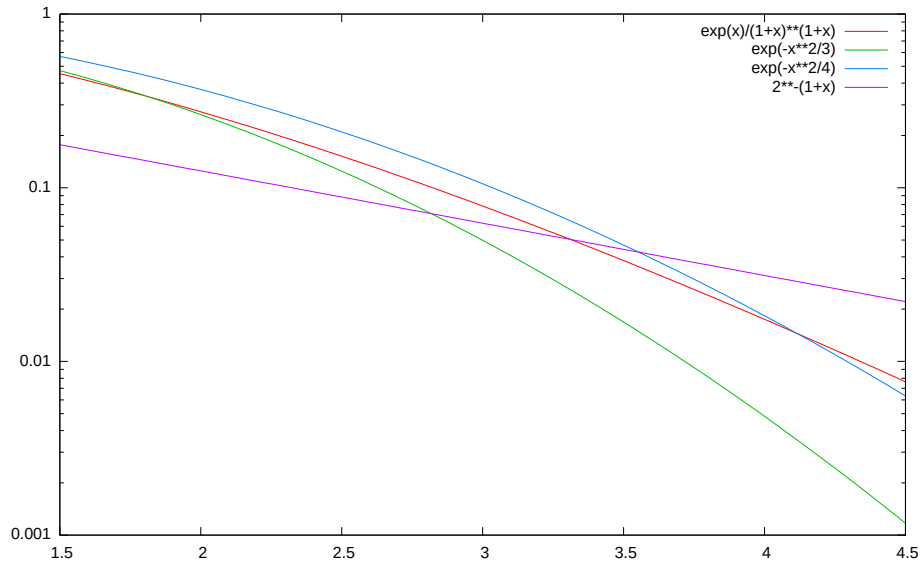


Figure 5.1: Comparison of Chernoff bound variants. The other bounds are valid only in the regions where they exceed $\exp(x)/(1+x)^{1+x}$.

5.2.3 Lower bound version

We can also use Chernoff bounds to show that a sum of independent 0–1 random variables isn't too small. The essential idea is to repeat the upper bound argument with a negative value of α, which makes $e^{\alpha(1-\delta)\mu}$ an increasing function in δ. The resulting bound is:

$$\Pr\left[S \leq (1-\delta)\mu\right] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu}. \tag{5.2.5}$$

A simpler but weaker version of this bound is

$$\Pr\left[S \leq (1-\delta)\mu\right] \leq e^{-\mu\delta^2/2}. \tag{5.2.6}$$

Both bounds hold for all δ with $0 \leq \delta \leq 1$.

5.2.4 Two-sided version

If we combine (5.2.2) with (5.2.6), we get

$$\Pr\left[|S-\mu| \geq \delta\mu\right] \leq 2e^{-\mu\delta^2/3}, \tag{5.2.7}$$

for $0 \leq \delta \leq 1.81$.

Suppose that we want this bound to be less than ϵ. Then we need $2e^{-\delta^2/3} \leq \epsilon$ or $\delta \geq \sqrt{\frac{3\ln(2/\epsilon)}{\mu}}$. Setting δ to exactly this quantity, (5.2.7) becomes

$$\Pr\left[|S-\mu| \geq \sqrt{3\mu\ln(2/\epsilon)}\right] \leq \epsilon, \tag{5.2.8}$$

provided $\epsilon \geq 2e^{-\mu/3}$.

For asymptotic purposes, we can omit the constants, giving

Lemma 5.2.1. *Let S be a sum of independent 0–1 variables with $\mathrm{E}[S] = \mu$. Then for any $0 < \epsilon \leq 2e^{-\mu/3}$, S lies within $O\left(\sqrt{\mu\log(1/\epsilon)}\right)$ of μ, with probability at least $1-\epsilon$.*

5.2.5 Other tail bounds for the binomial distribution

The random graph literature can be a good source for bounds on the binomial distribution. See for example [Bol01, §1.3], which uses normal approximation to get bounds that are slightly tighter than Chernoff bounds in some cases, and [JLR00, Chapter 2], which describes several variants of Chernoff bounds as well as tools for dealing with sums of random variables that aren't fully independent.

5.2.6 Applications

5.2.6.1 Flipping coins

Suppose S is the sum of n independent fair coin-flips. Then $\mathrm{E}[S] = n/2$ and $\Pr[S = n] = \Pr[S \geq 2\,\mathrm{E}[S]]$ is bounded using (5.2.1) by setting $\mu = n/2$, $\delta = 1$ to get $\Pr[S = n] \leq (e/4)^{n/2} = (2/\sqrt{e})^{-n}$. This is not quite as good as the real answer 2^{-n} (the quantity $2/\sqrt{e}$ is about 1.213...), but it's at least exponentially small.

5.2.6.2 Balls in bins again

Let's try applying the Chernoff bound to the balls-in-bins problem. Here we let $S = \sum_{i=1}^{m} X_i$ be the number of balls in a particular bin, with X_i the indicator that the i-th ball lands in the bin, $\mathrm{E}[X_i] = p_i = 1/n$, and $\mathrm{E}[S] = \mu = m/n$. To get a bound on $\Pr[S \geq m/n + k]$, apply the Chernoff

bound with $\delta = kn/m$ to get

$$\begin{aligned}\Pr[S \geq m/n + k] &= \Pr[S \geq (m/n)(1 + kn/m)] \\ &\leq \frac{e^k}{(1 + kn/m)^{1+kn/m}}.\end{aligned}$$

For $m = n$, this collapses to the somewhat nicer but still pretty horrifying $e^k/(k+1)^{k+1}$.

Staying with $m = n$, if we are bounding the probability of having large bins, we can use the 2^{-R} variant to show that the probability that any particular bin has more than $2 \lg n$ balls (for example), is at most n^{-2}, giving the probability that there exists such a bin of at most $1/n$. This is not as strong as what we can get out of the full Chernoff bound. If we take the logarithm of $e^k/(k+1)^{k+1}$, we get $k - (k+1)\ln(k+1)$; if we then substitute $k = \frac{c \ln n}{\ln \ln n} - 1$, we get

$$\begin{aligned}&\frac{c \ln n}{\ln \ln n} - 1 - \frac{c \ln n}{\ln \ln n} \ln \frac{c \ln n}{\ln \ln n} \\ &= (\ln n)\left(\frac{c}{\ln \ln n} - \frac{1}{\ln n} - \frac{c}{\ln \ln n}(\ln c + \ln \ln n - \ln \ln \ln n)\right) \\ &= (\ln n)\left(\frac{c}{\ln \ln n} - \frac{1}{\ln n} - \frac{c \ln c}{\ln \ln n} - c + \frac{c \ln \ln \ln n}{\ln \ln n}\right) \\ &= (\ln n)(-c + o(1)).\end{aligned}$$

So the probability of getting more than $c \ln n / \ln \ln n$ balls in any one bin is bounded by $\exp((\ln n)(-c + o(1))) = n^{-c+o(1)}$. This gives a maximum bin size of $O(\log n / \log \log n)$ with any fixed probability bound n^{-a} for sufficiently large n.

5.2.6.3 Flipping coins, central behavior

Suppose we flip n fair coins, and let S be the number that come up heads. We expect $\mu = n/2$ heads on average. How many extra heads can we get, if we want to stay within a probability bound of n^{-c}?

Here we use the small-δ approximation, which gives $\Pr[S \geq (1+\delta)(n/2)] \leq \exp(-\delta^2 n/6)$. Setting $\exp(-\delta^2 n/6) = n^{-c}$ gives $\delta = \sqrt{6 \ln n^c / n} = \sqrt{6c \ln n / n}$. The actual excess over the mean is $\delta(n/2) = (n/2)\sqrt{6c \ln n / n} = \sqrt{\frac{3}{2} cn \ln n}$. By symmetry, the same bound applies to extra tails. So if we flip 1000 coins and see more than 676 heads (roughly the bound when c=3), we can reasonably conclude that either (a) our coin is biased, or (b) we just hit a rare one-in-a-billion jackpot.

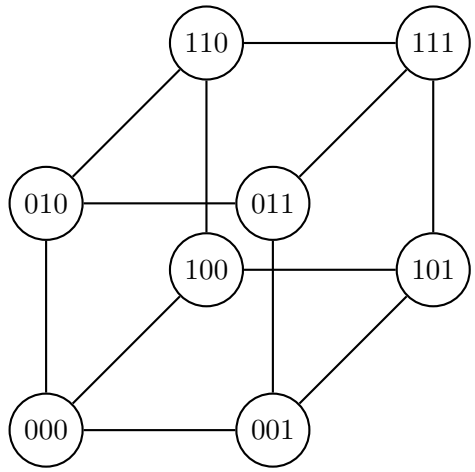


Figure 5.2: Hypercube network with $n = 3$

In algorithm analysis, the $\sqrt{(3/2)c}$ part usually gets absorbed into the asymptotic notation, and we just say that with probability at least $1 - n^{-c}$, the sum of n random bits is within $O(\sqrt{n \log n})$ of $n/2$.

5.2.6.4 Permutation routing on a hypercube

Here we use Chernoff bounds to show bounds on a classic **permutation-routing** algorithm for **hypercube networks** due to Valiant [Val82]. The presentation here is based on §§4.2 of [MR95], which in turn is based on an improved version of Valiant's original analysis appearing in a follow-up paper with Brebner [VB81]. There's also a write-up of this in [MU05, §4.5.1].

The basic idea of a hypercube architecture is that we have a collection of $N = 2^n$ processors, each with an n-bit address. Two nodes are adjacent if their addresses differ by one bit (see Figure 5.2 for an example). Though now mostly of theoretical interest, these things were the cat's pajamas back in the 1980s: see `http://en.wikipedia.org/wiki/Connection_Machine`.

Suppose that at some point in a computation, each processor i wants to send a packet of data to some processor $\pi(i)$, where π is a permutation of the addresses. But we can only send one packet per time unit along each of the n edges leaving a processor.[6] How do we route the packets so that all of them arrive in the minimum amount of time?

[6] Formally, we have a synchronous routing model with unbounded buffers at each node, with a maximum capacity of one packet per edge per round.

We could try to be smart about this, or we could use randomization. Valiant's idea is to first route each process i's packet to some random intermediate destination $\sigma(i)$, then in the second phase, we route it from $\sigma(i)$ to its ultimate destination $\pi(i)$. Unlike π, σ is not necessarily a permutation; instead, $\sigma(i)$ is chosen uniformly at random independently of all the other $\sigma(j)$. This makes the choice of paths for different packets independent of each other, which we will need later to apply Chernoff bounds..

Routing is done by a **bit-fixing**: if a packet is currently at node x and heading for node y, find the leftmost bit j where $x_j \neq y_j$ and fix it, by sending the packet on to $x[x_j/y_j]$. In the absence of contention, bit-fixing routes a packet to its destination in at most n steps. The hope is that the randomization will tend to spread the packets evenly across the network, reducing the contention for edges enough that the actual time will not be much more than this.

The first step is to argue that, during the first phase, any particular packet is delayed at most one time unit by any other packet whose path overlaps with it. Suppose packet i is delayed by contention on some edge uv. Then there must be some other packet j that crosses uv during this phase. From this point on, j remains one step ahead of i (until its path diverges), so it can't block i again unless both are blocked by some third packet k (in which case we charge i's further delay to k). This means that we can bound the delays for packet i by counting how many other packets cross its path.[7] So now we just need a high-probability bound on the number of packets that get in a particular packet's way.

Following the presentation in [MR95], define H_{ij} to be the indicator variable for the event that packets i and j cross paths during the first phase. Because each j chooses its destination independently, once we fix i's path, the H_{ij} are all independent. So we can bound $S = \sum_{j \neq i} H_{ij}$ using Chernoff bounds. To do so, we must first calculate an upper bound on $\mu = \mathrm{E}[S]$.

The trick here is to observe that any path that crosses i's path must cross one of its edges, and we can bound the number of such paths by bounding how many paths cross each edge. For each edge e, let T_e be the number of paths that cross edge e, and for each j, let X_j be the number of edges that path j crosses. Counting two ways, we have $\sum_e T_e = \sum_j X_j$, and so $\mathrm{E}\left[\sum_e T_e\right] = \mathrm{E}\left[\sum_j X_j\right] \leq N(n/2)$. By symmetry, all the T_e have the same expectation, so we get $\mathrm{E}[T_e] \leq \frac{N(n/2)}{Nn} = 1/2$.

Now fix $\sigma(i)$. This determines some path $e_1 e_2 \ldots e_k$ for packet i. In

[7] A much more formal version of this argument is given as [MR95, Lemma 4.5].

general we do not expect $\mathrm{E}[T_{e_\ell} \mid \sigma(i)]$ to equal $\mathrm{E}[T_{e_\ell}]$, because conditioning on i's path crossing e_ℓ guarantees that at least one path crosses this edge that might not have. However, if we let T'_e be the number of packets $j \neq i$ that cross e, then we have $T'_e \leq T_e$ always, giving $\mathrm{E}[T'_e] \leq \mathrm{E}[T_e]$, and because T'_e does not depend on i's path, $\mathrm{E}[T'_e \mid \sigma(i)] = \mathrm{E}[T'_e] \leq \mathrm{E}[T_e] \leq 1/2$. Summing this bound over all $k \leq n$ edges on i's path gives $\mathrm{E}[H_{ij} \mid \sigma(i)] \leq n/2$, which implies $\mathrm{E}[H_{ij}] \leq n/2$ after removing the conditioning on $\sigma(i)$.

Inequality (5.2.4) says that $\Pr[X \geq R] \leq 2^{-R}$ when $R \geq 2e\mu$. Letting $X = H_{ij}$ and setting $R = 3n$ gives $R = 6(n/2) \geq 6\mu > 2e\mu$, so $\Pr\left[\sum_j H_{ij} \geq 3n\right] \leq 2^{-3n} = N^{-3}$. This says that any one packet reaches its random destination with at most $3n$ added delay (thus, in at most $4n$ time units) with probability at least $1 - N^{-3}$. If we consider all N packets, the total probability that any of them fail to reach their random destinations in $4n$ time units is at most $N \cdot N^{-3} = N^{-2}$. Note that because we are using the union bound, we don't need independence for this step—which is good, because we don't have it.

What about the second phase? Here, routing the packets from the random destinations back to the real destinations is just the reverse of routing them from the real destinations to the random destinations. So the same bound applies, and with probability at most N^{-2} some packet takes more than $4n$ time units to get back (this assumes that we hold all the packets before sending them back out, so there are no collisions between packets from different phases).

Adding up the failure probabilities and costs for both stages gives a probability of at most $2/N^2$ that any packet takes more than $8n$ time units to reach its destination.

The structure of this argument is pretty typical for applications of Chernoff bounds: we get a very small bound on the probability that something bad happens by applying Chernoff bounds to a part of the problem where we have independence, then use the union bound to extend this to the full problem where we don't.

5.3 The Azuma-Hoeffding inequality

The problem with Chernoff bounds is that they only work for Bernoulli random variables. **Hoeffding's inequality** is another concentration bound based on the moment generating function that applies to any sum of bounded

independent random variables with mean 0.[8] It has the additional useful feature that it generalizes nicely to some collections of random variables that are not mutually independent, as we will see in §5.3.2. This more general version is known as **Azuma's inequality** or the **Azuma-Hoeffding inequality**.[9]

5.3.1 Hoeffding's inequality

This is the version for sums of bounded independent random variables.

Theorem 5.3.1. *Let $X_1 \dots X_n$ be independent random variables with $\mathrm{E}[X_i] = 0$ and $|X_i| \le c_i$ for all i. Then for all t,*

$$\Pr\left[\sum_{i=1}^{n} X_i \ge t\right] \le \exp\left(-\frac{t^2}{2\sum_{i=1}^{n} c_i^2}\right). \tag{5.3.1}$$

Proof. Let $S = \sum_{i=1}^{n} X_i$. As with Chernoff bounds, we'll first calculate a bound on the moment generating function $\mathrm{E}\left[e^{\alpha S}\right]$ and then apply Markov's inequality with a carefully-chosen α.

Before jumping into this, it helps to get a bound on $\mathrm{E}\left[e^{\alpha X}\right]$ when $\mathrm{E}[X] = 0$ and $|X| \le c$. The trick is that, for any α, $e^{\alpha x}$ is a convex function. Since we want an upper bound, we can't use Jensen's inequality (4.3.1), but we *can* use the fact that X is bounded and we know its expectation. Convexity of $e^{\alpha x}$ means that, for any x with $-c \le x \le c$, $e^{\alpha x} \le \lambda e^{-\alpha c} + (1-\lambda)e^{\alpha c}$, where $x = \lambda(-c) + (1-\lambda)c$. Solving for λ in terms of x gives $\lambda = \frac{1}{2}\left(1 - \frac{x}{c}\right)$ and $1 - \lambda = \frac{1}{2}\left(1 + \frac{x}{c}\right)$. So

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha X}\right] &\le \mathrm{E}\left[\frac{1}{2}\left(1 - \frac{X}{c}\right)e^{-\alpha c} + \frac{1}{2}\left(1 + \frac{X}{c}\right)e^{\alpha c}\right] \\
&= \frac{e^{-\alpha c} + e^{\alpha c}}{2} - \frac{e^{-\alpha c}}{2c}\mathrm{E}[X] + \frac{e^{\alpha c}}{2c}\mathrm{E}[X] \\
&= \frac{e^{-\alpha c} + e^{\alpha c}}{2} \\
&= \cosh(\alpha c).
\end{aligned}$$

[8] Note that the requirement that $\mathrm{E}[X_i] = 0$ can always be satisfied by considering instead $Y_i = X_i - \mathrm{E}[X_i]$.

[9] The history of this is that Hoeffding [Hoe63] proved it for independent random variables, and observed that the proof was easily extended to martingales, while Azuma [Azu67] actually went and did the work of proving it for martingales.

In other words, the worst possible X is a fair choice between $\pm c$, and in this case we get the hyperbolic cosine of αc as its moment generating function.

We don't like hyperbolic cosines much, because we are going to want to take products of our bounds, and hyperbolic cosines don't multiply very nicely. As before with $1 + x$, we'd be much happier if we could replace the cosh with a nice exponential. The Taylor's series expansion of $\cosh x$ starts with $1 + x^2/2 + \ldots$, suggesting that we should approximate it with $\exp(x^2/2)$, and indeed it is the case that for all x, $\cosh x \leq e^{x^2/2}$. This can be shown by comparing the rest of the Taylor's series expansions:

$$\begin{aligned}
\cosh x &= \frac{e^x + e^{-x}}{2} \\
&= \frac{1}{2}\left(\sum_{n=0}^{\infty} \frac{x^n}{n!} + \sum_{n=0}^{\infty} \frac{(-x)^n}{n!}\right) \\
&= \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!} \\
&\leq \sum_{n=0}^{\infty} \frac{x^{2n}}{2^n n!} \\
&= \sum_{n=0}^{\infty} \frac{(x^2/2)^n}{n!} \\
&= e^{x^2/2}.
\end{aligned}$$

This gives $\mathrm{E}\left[e^{\alpha X}\right] \leq e^{(\alpha c)^2/2}$.

Using this bound and the independence of the X_i, we compute

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha S}\right] &= \mathrm{E}\left[\exp\left(\alpha \sum_{i=1}^{n} X_i\right)\right] \\
&= \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha X_i}\right] \\
&= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha X_i}\right]. \\
&\leq \prod_{i=1}^{n} e^{(\alpha c_i)^2/2} \\
&= \exp\left(\sum_{i=1}^{n} \frac{\alpha^2 c_i^2}{2}\right) \\
&= \exp\left(\frac{\alpha^2}{2} \sum_{i=1}^{n} c_i^2\right).
\end{aligned}$$

Applying Markov's inequality then gives (when $\alpha > 0$):

$$\begin{aligned}
\Pr\left[S \geq t\right] &= \Pr\left[e^{\alpha S} \geq e^{\alpha t}\right] \\
&\leq \exp\left(\frac{\alpha^2}{2} \sum_{i=1}^{n} c_i^2 - \alpha t\right). \qquad (5.3.2)
\end{aligned}$$

Now we do the same trick as in Chernoff bounds and choose α to minimize the bound. If we write C for $\sum_{i=1}^{n} c_i^2$, this is done by minimizing the exponent $\frac{\alpha^2}{2}C - \alpha t$, which we do by taking the derivative with respect to α and setting it to zero: $\alpha C - t = 0$, or $\alpha = t/C$. At this point, the exponent becomes $\frac{(t/C)^2}{2}C - (t/C)t = -\frac{t^2}{2C}$.

Plugging this into (5.3.2) gives the bound (5.3.1) claimed in the theorem. □

Let's see how good a bound this gets us for our usual test problem of bounding $\Pr\left[S = n\right]$ where $S = \sum_{i=1}^{n} X_i$ is the sum of n independent fair coin-flips. To make the problem fit the theorem, we replace each X_i by a rescaled version $Y_i = 2X_i - 1 = \pm 1$ with equal probability; this makes $\mathrm{E}\left[Y_i\right] = 0$ as needed, with $|Y_i| \leq c_i = 1$. Hoeffding's inequality (5.3.1) then

gives

$$\Pr\left[\sum_{i=1}^{n} Y_i \geq n\right] \leq \exp\left(-\frac{n^2}{2n}\right)$$
$$= e^{-n/2} = (\sqrt{e})^{-n}.$$

Since $\sqrt{e} \approx 1.649\ldots$, this is actually slightly better than the $(2/\sqrt{e})^{-n}$ bound we get using Chernoff bounds.

On the other hand, Chernoff bounds work better if we have a more skewed distribution on the X_i; for example, in the balls-in-bins case, each X_i is a Bernoulli random variable with $\mathrm{E}[X_i] = 1/n$. Using Hoeffding's inequality, we get a bound c_i on $|X_i - \mathrm{E}[X_i]|$ of only $1 - 1/n$, which puts $\sum_{i=1}^{n} c_i^2$ very close to n, requiring $t = \Omega(\sqrt{n})$ before we get any non-trivial bound out of (5.3.1), pretty much the same as in the fair-coin case (which is not surprising, since Hoeffding's inequality doesn't know anything about the distribution of the X_i). But we've already seen that Chernoff gives us that $\sum X_i = O(\log n/\log\log n)$ with high probability in this case.

Note: There is an asymmetrical version of Hoeffding's inequality in which $a_i \leq X_i \leq b_i$, but $\mathrm{E}[X_i]$ is still zero for all X_i. In this version, the bound is

$$\Pr\left[\sum_{i=1}^{n} X_i \geq t\right] \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\right). \tag{5.3.3}$$

This reduces to (5.3.1) when $a_i = -c_i$ and $b_i = c_i$. The proof is essentially the same, but a little more analytic sneakery is required to show that $\mathrm{E}\left[e^{\alpha X_i}\right] \leq e^{\alpha^2(b_i-a_i)^2/8}$; see [McD89] for details. For most applications, the only difference between the symmetric version (5.3.1) and the asymmetric version (5.3.3) is a small constant factor on the resulting bound on t.

Hoeffding's inequality as stated above is not the tightest possible, but is relatively simple and easy to work with. For a particularly strong version of Hoeffding's inequality and a discussion of some variants, see [FGQ12].

5.3.2 Azuma's inequality

A general rule of thumb is that most things that work for sums of independent random variables also work for martingales, which are similar collections of random variables that allow for more dependence.

Formally, a **martingale** is a sequence of random variables $S_0, S_1, S_2, \ldots$, where $\mathrm{E}[S_t \mid S_1, \ldots, S_{t-1}] = S_{t-1}$. In other words, given everything you know up until time $t-1$, your best guess of the expected value at time t is just wherever you are now.

Another way to describe a martingale is to take the partial sums $S_t = \sum_{i=1}^{t} X_t$ of a **martingale difference sequence**, which is a sequence of random variables $X_1, X_2, \ldots$ where $\mathrm{E}[X_t \mid X_1 \ldots X_{t-1}] = 0$. So in this version, your expected change from time $t-1$ to t averages out to zero, even if you try to predict it using all the information you have at time $t-1$.

Martingales were invented to analyze fair gambling games, where your return over some time interval is not independent of previous outcomes (for example, you may change your bet or what game you are playing depending on how things have been going for you), but it is always zero on average given previous information (warning: real casinos give negative expected return, so the resulting process is a **supermartingale** with $S_{t-1} \geq \mathrm{E}[S_t \mid S_0 \ldots S_{t-1}]$). The nice thing about martingales is they allow for a bit of dependence while still acting very much like sums of independent random variables.

Where this comes up with Hoeffding's inequality is that we might have a process that is reasonably well-behaved, but its increments are not technically independent. For example, suppose that a gambler plays a game where she bets x units $0 \leq x \leq 1$ at each round, and receives $\pm x$ with equal probability. Suppose also that her bet at each round may depend on the outcome of previous rounds (for example, she might stop betting entirely if she loses too much money). If X_i is her take at round i, we have that $\mathrm{E}[X_i \mid X_1 \ldots X_{i-1}] = 0$ and that $|X_i| \leq 1$. This is enough to apply the martingale version of Hoeffding's inequality, often called Azuma's inequality.

Theorem 5.3.2. *Let $\{S_k\}$ be a martingale with $S_k = \sum_{i=1}^{k} X_i$ and $|X_i| \leq c_i$ for all i. Then for all n and all $t \geq 0$:*

$$\Pr[S_n \geq t] \leq \exp\left(\frac{-t^2}{2\sum_{i=1}^{n} c_i^2}\right). \tag{5.3.4}$$

Proof. Basically, we just show that $\mathrm{E}\left[e^{\alpha S_n}\right] \leq \exp\left(\frac{\alpha^2}{2}\sum_{i=1}^{n} c_i^2\right)$—just like in the proof of Theorem 5.3.1—and the rest follows using the same argument. The only tricky part is we can no longer use independence to transform $\mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha X_i}\right]$ into $\prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha X_i}\right]$.

Instead, we use the martingale property. For each X_i, we have $\mathrm{E}[X_i \mid X_1 \ldots X_{i-1}] = 0$ and $|X_i| \leq c_i$ always. Recall that $\mathrm{E}\left[e^{\alpha X_i} \,\middle|\, X_1 \ldots X_{i-1}\right]$ is a random variable that takes on the average value of $e^{\alpha X_i}$ for each setting of $X_1 \ldots X_{i-1}$. We can apply the same analysis as in the proof of 5.3.1 to show that this means that $\mathrm{E}\left[e^{\alpha X_i} \,\middle|\, X_1 \ldots X_{i-1}\right] \leq e^{(\alpha c_i)^2/2}$ always.

The trick is to use the fact that, for any random variables X and Y, $\mathrm{E}[XY] = \mathrm{E}[\mathrm{E}[XY \mid X]] = \mathrm{E}[X\,\mathrm{E}[Y \mid X]]$.

We argue by induction on n that $\mathrm{E}\left[\prod_{i=1}^n e^{\alpha X_i}\right] \leq \prod_{i=1}^n e^{(\alpha c)^2/2}$. The base case is when $n = 0$. For the induction step, compute

$$\begin{aligned}
\mathrm{E}\left[\prod_{i=1}^n e^{\alpha X_i}\right] &= \mathrm{E}\left[\mathrm{E}\left[\prod_{i=1}^n e^{\alpha X_i} \;\middle|\; X_1 \dots X_{n-1}\right]\right] \\
&= \mathrm{E}\left[\left(\prod_{i=1}^n e^{\alpha X_i}\right) \mathrm{E}\left[e^{\alpha X_n} \;\middle|\; X_1 \dots X_{n-1}\right]\right] \\
&\leq \mathrm{E}\left[\left(\prod_{i=1}^{n-1} e^{\alpha X_i}\right) e^{(\alpha c_n)^2/2}\right] \\
&= \mathrm{E}\left[\prod_{i=1}^{n-1} e^{\alpha X_i}\right] e^{(\alpha c_n)^2/2} \\
&\leq \left(\prod_{i=1}^{n-1} e^{(\alpha c_i)^2/2}\right) e^{(\alpha c_n)^2/2} \\
&= \prod_{i=1}^n e^{(\alpha c_i)^2/2}. \\
&= \exp\left(\frac{\alpha^2}{2} \sum_{i=1}^n c_i^2\right).
\end{aligned}$$

The rest of the proof goes through as before. □

Some extensions:

- The asymmetric version of Hoeffding's inequality (5.3.3) also holds for martingales. So if each increment X_i satisfies $a_i \leq X_i \leq b_i$ always,

$$\Pr\left[\sum_{i=1}^n X_i \geq t\right] \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^n (b_i - a_i)^2}\right). \tag{5.3.5}$$

- The same bound works for bounded-difference **supermartingales**. A supermartingale is a process where $\mathrm{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$; the idea is that my expected gain at any step is non-positive, so my present wealth is always superior to my future wealth.[10] If $\mathrm{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$ and $|X_i| \leq c_i$, then we can write $X_i = Y_i + Z_i$ where $Y_i = \mathrm{E}[X_i \mid X_1 \dots X_{i-1}] \leq 0$ is predictable from $X_1 \dots X_{i-1}$ and $\mathrm{E}[Z_i \mid X_1 \dots X_{i-1}] = 0$.[11] Then we can bound $\sum_{i=1}^n X_i$ by observing that it is no greater than $\sum_{i=1}^n Z_i$.

[10] The corresponding notion in the other direction is a **submartingale**. See §8.1.

[11] This is known as a **Doob decomposition** and can be used to extract a martingale $\{Z_i\}$ from any stochastic process $\{X_i\}$. For general processes, $Y_i = X_i - Z_i$ will still be predictable, but may not satisfy $Y_i \leq 0$.

A complication is that we no longer have $|Z_i| \leq c_i$; instead, $|Z_i| \leq 2c_i$ (since leaving out Y_i may shift Z_i up). But with this revised bound, (5.3.4) gives

$$\begin{aligned}\Pr\left[\sum_{i=1}^{n} X_i \geq t\right] &\leq \Pr\left[\sum_{i=1}^{n} Z_i \geq t\right] \\ &\leq \exp\left(-\frac{t^2}{8\sum_{i=1}^{n} c_i^2}\right). \end{aligned} \tag{5.3.6}$$

- Suppose that we stop the process after the first time τ with $S_\tau = \sum_{i=1}^{\tau} X_i \geq t$. This is equivalent to making a new variable Y_i that is zero whenever $S_{i-1} \geq t$ and equal to X_i otherwise. This doesn't affect the conditions $\mathrm{E}\left[Y_i \mid Y_1 \ldots Y_{i-1}\right] = 0$ or $|Y_i| \leq c_i$, but it makes it so $\sum_{i=1}^{n} Y_i \geq t$ if and only if $\max_{k \leq n} \sum_{i=1}^{k} X_i \geq t$. Applying (5.3.4) to $\sum Y_i$ then gives

$$\Pr\left[\max_{k \leq n} \sum_{i=1}^{k} X_i \geq t\right] \leq \exp\left(-\frac{t^2}{2\sum_{i=1}^{n} c_i^2}\right). \tag{5.3.7}$$

- Since the conditions on X_i in Theorem 5.3.2 apply equally well to $-X_i$, we have

$$\Pr\left[\sum_{i=1}^{n} X_i \leq -t\right] \leq \exp\left(-\frac{t^2}{2\sum_{i=1}^{n} c_i^2}\right). \tag{5.3.8}$$

  which we can combine with (5.3.4) to get the two-sided bound

$$\Pr\left[\left|\sum_{i=1}^{n} X_i\right| \geq t\right] \leq 2\exp\left(-\frac{t^2}{2\sum_{i=1}^{n} c_i^2}\right). \tag{5.3.9}$$

- The extension of Hoeffding's inequality to the case $a_i \leq X_i \leq b_i$ works equally well for Azuma's inequality, giving the same bound as in (5.3.3).

- Finally, one can replace the requirement that each c_i be a constant with a requirement that c_i be predictable from $X_1 \ldots X_{i-1}$ and that $\sum_{i=1}^{n} c_i^2 \leq C$ always and get $\Pr\left[\sum_{i=1}^{n} X_i \geq t\right] \leq e^{-t^2/2C}$. This generally doesn't come up unless you have an algorithm that explicitly cuts off the process if $\sum c_i^2$ gets too big, but there is at least one example of this in the literature [AW96].

5.3.3 The method of bounded differences

To use Azuma's inequality, we need a bounded-difference martingale. The easiest way to get such martingales is through the **method of bounded differences**, which was popularized by a survey paper by McDiarmid [McD89]. For this reason the key result is often referred to as **McDiarmid's inequality**.

The basic idea of the method is to imagine we are computing a function $f(X_1, \ldots, X_n)$ of a sequence of independent random variables $X_1, \ldots, X_n$. To get our martingale, we'll imagine we reveal the X_i one at a time, and compute at each step the expectation of the final value of f based on just the inputs we've seen so far.

Formally, let $Y_t = \mathrm{E}[f \mid X_1, \ldots, X_t]$, the expected value of f given the values of the first t variables. Then $\{Y_t\}$ forms a martingale, with $Y_0 = \mathrm{E}[f]$ and $Y_n = \mathrm{E}[f \mid X_1, \ldots, X_t] = f$.[12] So if we can find a bound c_t on $Y_t - Y_{t-1}$, we can apply Azuma's inequality to get bounds on $Y_n - Y_0 = f - \mathrm{E}[f]$.

We do this by assuming that f is **Lipschitz** with respect to **Hamming distance** on its domain.[13] This means that there are bounds c_t such that for any $x_1 \ldots x_n$ and any x'_t, we have

$$\left|f(x_1 \ldots x_t \ldots x_n) - f(x_1 \ldots x'_t \ldots x_n)\right| \leq c_t. \tag{5.3.10}$$

If f has this property, the following result holds:

Theorem 5.3.3 (**McDiarmid's inequality** [McD89])**.** *Let $X_1, \ldots, X_n$ be independent random variables and let $f(X_1, \ldots, X_n)$ be Lipschitz with bounds*

[12] A sequence of random variables of the form $Y_t = \mathrm{E}[Z \mid X_1, \ldots, X_t]$ is called a **Doob martingale**. The proof that it is a martingale is slightly painful using the tools we've got, but the basic idea is that we can expand $\mathrm{E}[Y_t \mid X_1, \ldots, X_{t-1}] = \mathrm{E}[\mathrm{E}[Z \mid X_1, \ldots, X_t] \mid X_1, \ldots, X_{t-1}] = \mathrm{E}[Z \mid X_1, \ldots, X_{t-1}] = Y_{t-1}$, where the step in the middle follows from the usual repeated-averaging trick that shows $\mathrm{E}[\mathrm{E}[X \mid Y, Z] \mid Z] = \mathrm{E}[X \mid Z]$. To change the variables we are conditioning Y_t on from X_i's to Y_i's, we have to observe that the X_i give at least as much information as the corresponding Y_i (since we can calculate $Y_1, \ldots, Y_{t-1}$ from $X_1, \ldots, X_{t-1}$), so $\mathrm{E}[Y_t \mid Y_1, \ldots, Y_{t-1}] = \mathrm{E}[\mathrm{E}[Y_t \mid X_1, \ldots, X_{t-1}] \mid Y_1, \ldots, Y_{t-1}] = \mathrm{E}[Y_{t-1} \mid Y_1, \ldots, Y_{t-1}] = Y_{t-1}$. This establishes the martingale property.

[13] The Hamming distance between two vectors is the number of places where they differ, without regard to how much they differ by in these places. A Lipschitz function in general is a function f such that there exists a constant c for which $d(f(x), f(y)) \leq c \cdot d(x, y)$ for all x, y in f's domain, where d is the distance between two points in our preferred metrics for the domain and codomain. This is similar to but stronger than the usual notion of continuity.

c_i. *Then*

$$\Pr\left[f(X_1,\ldots,X_n) - \mathrm{E}\left[f(X_1,\ldots,X_n)\right] \geq t\right] \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^n c_i^2}\right). \quad (5.3.11)$$

Since $-f$ satisfies the same constraints as f, the same bound holds for $\Pr\left[f(X_1,\ldots,X_n) - \mathrm{E}\left[f(X_1,\ldots,X_n)\right] \leq -t\right]$.

We will not prove Theorem 5.3.3, as getting the exact constant in the exponent is tricky; see [McD89] for a proof. However, if we are willing to accept a weaker bound, we can easily show that if f satisfies the conditions of the theorem, then revealing the i-th value of X_i changes the conditional expectation of $f(X_1,\ldots,X_n)$ by at most c_i. This gives a bound of $\exp\left(-\frac{t^2}{2\sum_{i=1}^n c_i^2}\right)$ from Azuma's inequality. Note that (5.3.11) puts the 2 in the numerator instead of the denominator of the exponent, giving a stronger bound.

5.3.4 Applications

Here are some applications of the preceding inequalities. Most of these are examples of the method of bounded differences.

5.3.4.1 Sprinkling points on a hypercube

Suppose you live in a hypercube, and the local government has conveniently placed snow shovels on some subset A of the nodes. If you start at a random location, how likely is it that your distance to the nearest snow shovel deviates substantially from the average distance?

We can describe your position as a bit vector $X_1,\ldots,X_n$, where each X_i is an independent random bit. Let $f(X_1,\ldots,X_n)$ be the distance from $X_1,\ldots,X_n$ to the nearest element of A. Then changing one of the bits changes this function by at most 1. So we have $\Pr\left[|f - \mathrm{E}\left[f\right]| \geq t\right] \leq 2e^{-2t^2/n}$ by (5.3.11), giving a range of possible distances that is $O(\sqrt{n\log n})$ with probability at least $1 - n^{-c}$ for any fixed $c > 0$.[14] Of course, without knowing what A is, we don't know what $E[f]$ is; but at least we can be assured that (unless A is very big) the distance we have to walk through the snow will be pretty much the same pretty much wherever we start.

[14]Proof: Let $t = \sqrt{\frac{1}{2}(c+1)n\ln n} = O(\sqrt{n\log n})$. Then $2e^{-2t^2/n} = 2e^{-(c+1)\ln n} = 2n^{-c-1} < n^{-c}$ when n is sufficiently large.

5.3.4.2 Chromatic number of a random graph

Consider a **random graph** $G(n, p)$ consisting of n vertices, where each possible edge appears with independent probability p. Let χ be the **chromatic number** of this graph, the minimum number of colors necessary if we want to assign a color to each vertex that is distinct for the colors of all of its neighbors. The **vertex exposure martingale** shows us the vertices of the graph one at a time, along with all the edges between vertices that have been exposed so far. We define X_t to be the expected value of χ given this information for vertices $1 \ldots t$.

If Z_i is a random variable describing which edges are present between i and vertices less than i, then the Z_i are all independent, and we can write $\chi = f(Z_1, \ldots, Z_n)$ for some function f (this function may not be very easy to compute, but it exists). Then X_t as defined above is just $\mathrm{E}[f \mid Z_1, \ldots, Z_t]$. Now observe that f is Lipschitz with $c_t = 1$: if I change the edges for some vertex v_t, I can't increase the number of colors I need by more than 1, since in the worst case I can always take whatever coloring I previously had for all the other vertices and add a new color for v_t. (This implies I can't decrease χ by more than one either.) McDiarmid's inequality (5.3.11) then says that $\Pr[|\chi - \mathrm{E}[\chi]| \geq t] \leq 2e^{-2t^2/n}$; in other words, the chromatic number of a random graph is tightly concentrated around its mean, even if we don't know what that mean is. (This proof is due to Shamir and Spencer [SS87].)

5.3.4.3 Balls in bins

Suppose we toss m balls into n bins. How many empty bins do we get? The probability that each bin individually is empty is exactly $(1 - 1/n)^m$, which is approximately $e^{-m/n}$ when n is large. So the expected number of empty bins is exactly $n(1 - 1/n)^m$. If we let X_i be the bin that ball i gets tossed into, and let $Y = f(X_1, \ldots, X_m)$ be the number of empty bins, then changing a single X_i can change f by at most 1. So from (5.3.11) we have $\Pr[Y \geq n(1 - 1/n)^m + t] \leq e^{-2t^2/m}$.

5.3.4.4 Probabilistic recurrence relations

Most probabilistic recurrence arguments (as in Appendix E) can be interpreted as supermartingales: the current estimate of $T(n)$ is always at least the expected estimate after doing one stage of the recurrence. This fact can be used to get concentration bounds using (5.3.6).

For example, let's take the recurrence (1.3.1) for the expected number

of comparisons for QuickSort:

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left(T(k) + T(n-1-k)\right).$$

We showed in §1.3.1 that the solution to this recurrence satisfies $T(n) \leq 2n \ln n$.

To turn this into a supermartingale, imagine that we carry out a process where we keep around at each step t a set of unsorted blocks of size $n_1^t, n_2^t, \ldots, n_{k_t}^t$ for some k_t (note that the superscripts on n_i^t are not exponents). One step of the process involves choosing one of the blocks (we can do this arbitrarily without affecting the argument) and then splitting that block around a uniformly-chosen pivot. We will track a random variable X_t equal to $C_t + \sum_{i=1}^{k_t} 2n_i^t \ln n_i^t$, where C_t is the number of comparisons done so far and the summation gives an upper bound on the expected number of comparisons remaining.

To show that this is in fact a supermartingale, observe that if we partition a block of size n we add n to C_t but replace the cost bound $2n \ln n$ by an expected

$$\begin{aligned}
2 \cdot \frac{1}{n}\sum_{k=0}^{n-1} 2k \ln k &\leq \frac{4}{n}\int_2^n n \ln n \\
&= \frac{4}{n}\left(\frac{n^2 \ln n}{2} - \frac{n^2}{4} - \ln 2 + 1\right) \\
&= 2n \ln n - n - \ln 2 + 1 \\
&< 2n \ln n - n.
\end{aligned}$$

The net change is less than $-\ln 2$. The fact that it's not zero suggests that we could improve the $2n \ln n$ bound slightly, but since it's going down, we have a supermartingale.

Let's try to get a bound on how much X_t changes at each step. The C_t part goes up by at most $n-1$. The summation can only go down; if we split a block of size n_i, the biggest drop we get is if we split it evenly,[15] This

[15] This can be proven most easily using convexity of $n \ln n$.

gives a drop of

$$\begin{aligned}
2n\ln n - 2\left(2\frac{n-1}{2}\ln\frac{n-1}{2}\right) &= 2n\ln n - 2(n-1)\ln\left(n\frac{n-1}{2n}\right) \\
&= 2n\ln n - 2(n-1)(\ln n - \ln\frac{2n}{n-1}) \\
&= 2n\ln n - 2n\ln n + 2n\ln\frac{2n}{n-1} + 2\ln n - 2\ln\frac{2n}{n-1} \\
&= 2n\cdot O(1) + O(\log n) \\
&= O(n).
\end{aligned}$$

(with a constant tending to 2 in the limit).

So we can apply (5.3.6) with $c_t = O(n)$ to the at most n steps of the algorithm, and get

$$\Pr\left[C_n - 2n\ln n \geq t\right] \leq e^{-t^2/O(n^3)}.$$

This gives $C_n = O(n^{3/2})$ with constant probability or $O(n^{3/2}\sqrt{\log n})$ with all but polynomial probability. This is a rather terrible bound, but it's a lot better than $O(n^2)$. For a much better bound, see [MH92].

5.3.4.5 Multi-armed bandits

In the **multi-armed bandit** problem, we must choose at each time step one of a fixed set of k **arms** to pull. Pulling arm i at time t yields a return of X_i^t, a random payoff typically assumed to be between 0 and 1. Suppose that all the X_i^t are independent, and that for each fixed i, all X_i^t have the same distribution, and thus the same expected payoff. Suppose also that we initially know nothing about these distributions. What strategy can we use to maximize our expected payoff over a large number of pulls?

More specifically, how can we minimize our **regret**, defined as

$$\mu^* - \sum_{i=1}^{k} \mathrm{E}\left[X_i\right] T_i, \tag{5.3.12}$$

where μ^* is the expected payoff of the best arm, and T_i counts the number of times we pull arm i?

The tricky part here is that when we pull an arm and get a bad return, we don't know if we were just unlucky this time or it's actually a bad arm. So we have an incentive to try lots of different arms. On the other hand, the more we pull a genuinely inferior arm, the worse our overall return. We'd

like to adopt a strategy that trades off between exploration (trying new arms) and exploitation (collecting on the best arm so far) to do as best we can in comparison to a strategy that always pulls the best arm.

The UCB1 algorithm Fortunately, there is a simple algorithm due to Auer *et al.* [ACBF02] that solves this problem for us.[16] To start with, we pull each arm once. For any subsequent pull, suppose that for each i, we have pulled the i-th arm n_i times so far. Let $n = \sum n_i$, and let $\overline{x}_i$ be the average payoff from arm i so far. Then the **UCB1** algorithm pulls the arm that maximizes

$$\overline{x}_i + \sqrt{\frac{2 \ln n}{n_i}}. \tag{5.3.13}$$

UCB stands for **upper confidence bound**. (The "1" is because it is the first, and simplest, of several algorithms of this general structure given in the paper.) The idea is that we give arms that we haven't tried a lot the benefit of the doubt, and assume that their actual average payout lies at the upper end of some plausible range of values.[17]

The quantity $\sqrt{\frac{2 \ln n}{n_i}}$ is a bit mysterious, but it arises in a fairly natural way from the asymmetric version of Hoeffding's inequality. With a small adjustment to deal with non-zero-mean variables, (5.3.3) says that, if S is a sum of n random variables bounded between a_i and b_i, then

$$\Pr\left[\sum_{i=1}^{n} (\mathrm{E}[X_i] - X_i) \geq t\right] \leq e^{-2t^2/\sum_{i=1}^{n}(b_i - a_i)^2}. \tag{5.3.14}$$

Now consider $\overline{x}_i = \sum_{j=1}^{n_i} X_j$ where each X_j lies between 0 and $1/n_i$. Then (5.3.14) says that

$$\begin{aligned} \Pr\left[\mathrm{E}[\overline{x}_i] - \overline{x}_i \geq \sqrt{\frac{2 \ln n}{n_i}}\right] &\leq e^{-2(\sqrt{(2 \ln n)/n_i})^2/(n_i(1/n_i)^2)} \\ &= e^{-4 \ln n} \\ &= n^{-4}. \end{aligned} \tag{5.3.15}$$

[16] This is not the only algorithm for solving multi-armed bandit problems, and it's not even the only algorithm in the Auer *et al.* paper. But it has the advantage of being relatively straightforward to analyze. For a more general survey of multi-armed bandit algorithms for the regret model, see [BCB12].

[17] The "bound" part is because we don't attempt to compute the exact upper end of this confidence interval, which may be difficult, but instead use an upper bound derived from Hoeffding's inequality. This distinguishes the UCB1 algorithm of [ACBF02] from the *upper confidence interval* approach of Lai and Robbins [LR85] that it builds on.

So the bonus applied to $\overline{x}_i$ is really a high-probability bound on how big the difference between the observed payoff and expected payoff might be. The $\sqrt{\lg n}$ bit is there to make the error probability be small as a function of n, since we will be summing over a number of bad cases polynomial in n and not a particular n_i.

In terms of actual execution of the algorithm, even a bad arm will be pulled infinitely often, as $\ln n$ rises enough to compensate for the last increase in n_i. This accounts for an $O(\log n)$ term in the regret, as we will see below.

Analysis of UCB1 The following theorem is a direct quote of [ACBF02, Theorem 1]:

Theorem 5.3.4 ([ACBF02])**.** *For all $K > 1$, if policy UCB1 is run on K machines having arbitrary reward distributions $P_1, \ldots, P_K$ with support in $[0, 1]$, then its expected regret after any number n of plays is at most*

$$\left[8 \sum_{i:\mu_i<\mu^*} \frac{\ln n}{\Delta_i}\right] + \left(1 + \frac{\pi^2}{3}\right)\left(\sum_{j=1}^{K} \Delta_j\right). \tag{5.3.16}$$

Here $\mu_i = \mathrm{E}[P_i]$ is the expected payoff for arm i, μ^* as before is $\max_i \mu_i$, and $\Delta_i = \mu^* - \mu_i$ is the regret for pulling arm i. The theorem states that our expected regret is bounded by a term for each arm worse that μ^* that grows logarithmically in n and is inversely proportional to how close the arm is to optimal, plus a constant additional loss corresponding to pulling every arm a little more than 4 times on average. The logarithmic regret in n is a bit of a nuisance, but an earlier lower bound of Lai and Robbins [LR85] shows that something like this is necessary in the limit.

To prove Theorem 5.3.4, we need to get an upper bound on the number of times each suboptimal arm is pulled during the first n pulls. Define

$$c_{t,s} = \sqrt{\frac{2 \ln t}{s}}, \tag{5.3.17}$$

the bonus given to an arm that has been pulled s times in the first t pulls.

Fix some optimal arm. Let $\overline{X}_{i,s}$ be the average return on arm i after s pulls and $\overline{X}^*_s$ be the average return on this optimal arm after s pulls.

If we pull arm i after t total pulls, when arm i has previously been pulled s_i times and our optimal arm has been pulled s^* times, then we must have

$$\overline{X}_{i,s_i} + c_{t,s_i} \geq \overline{X}^*_{s^*} + C_{t,s^*}. \tag{5.3.18}$$

This just says that arm i with its bonus looks better than the optimal arm with its bonus.

To show that this bad event doesn't happen, we need three things:

1. The value of $\overline{X}^* s^* + c_{t,s^*}$ should be at least μ^*. Were it to be smaller, the observed value $\overline{X}^* s^*$ would be more than C_{t,s^*} away from its expectation Hoeffding's inequality implies this doesn't happen too often.

2. The value of $\overline{X}_{i,s_i} + c_{t,s_i}$ should not be too much bigger than μ_i. We'll again use Hoeffding's inequality to show that $\overline{X}_{i,s_i}$ is likely to be at most $\mu_i + c_{t,s_i}$, making $\overline{X}_{i,s_i} + c_{t,s_i}$ at most $\mu_i + 2c_{t,s_i}$.

3. The bonus c_{t,s_i} should be small enough that even adding $2c_{t,s_i}$ to μ_i is not enough to beat μ^*. This means that we need to pick s_i large enough that $2c_{t,s_i} \leq \Delta_i$. For smaller values of s_i, we will just accept that we need to pull arm i a few more times before we wise up.

More formally, if none of the following events hold:

$$\overline{X}^*_{s^*} + c_{t,s^*} \leq \mu^*. \tag{5.3.19}$$

$$\overline{X}_{i,s_i} \geq \mu_i + c_{t,s_i} \tag{5.3.20}$$

$$\mu^* - \mu_i < 2c_{t,s_i}, \tag{5.3.21}$$

then $\overline{X}^*_{s^*} + c_{t,s^*} > \mu^* > \mu_i + 2c_{t,s_i} > \overline{X}_{i,s_i}$, and we don't pull arm i because the optimal arm is better. (We don't necessarily pull the optimal arm, but if we don't, it's because we pull some other arm that still isn't arm i.)

For (5.3.19) and (5.3.20), we repeat the argument in (5.3.15), plugging in t for n and s_i or s^* for n_i. This gives a probability of at most $2t^{-4}$ that either or both of these bad events occur.

For (5.3.21), we need to do something a little sneaky, because the statement is not actually true when s_i is small. So we will give ℓ free pulls to arm i, and only start comparing arm i to the optimal arm after we have done at least this many pulls. The value of ℓ is chosen so that, when $t \leq n$ and $s_i > \ell$,

$$2c_{t,s_i} \leq \mu^* - \mu_i,$$

which expands to,

$$2\sqrt{\frac{2\ln t}{s_i}} \leq \Delta_i,$$

giving

$$s_i \geq \frac{8 \ln t}{\Delta_i^2}.$$

So we must set ℓ to be at least

$$\frac{8 \ln n}{\Delta_i^2} \geq \frac{8 \ln t}{\Delta_i^2}.$$

Because ℓ must be an integer, we actually get

$$\ell = \left\lceil \frac{8 \ln n}{\Delta_i^2} \right\rceil \leq 1 + \frac{8 \ln n}{\Delta_i^2}.$$

This explains (after multiplying by the regret Δ_i) the first term in (5.3.16).

For the other sources of bad pulls, apply the union bound to the $2t^{-4}$ error probabilities we previously computed for all choices of $t \leq n$, $s^* \geq 1$, and $s_i > \ell$. This gives

$$\begin{aligned} \sum_{t=1}^{n} \sum_{s^*=1}^{t-1} \sum_{s_i=\ell+1}^{t-1} 2t^{-4} &< 2 \sum_{t=1}^{\infty} t^2 \cdot t^{-4} \\ &= 2 \cdot \frac{\pi^2}{6} \\ &= \frac{\pi^2}{3}. \end{aligned}$$

Again we have to multiply by the regret Δ_i for pulling the i-th arm, which gives the second term in (5.3.16).

5.4 Anti-concentration bounds

It may be that for some problem you want to show that a sum of random variables is far from its mean at least some of the time: this would be an **anti-concentration bound**. Anti-concentration bounds are much less well-understood than concentration bounds, but there are known results that can help in some cases.

For variables where we know the distribution of the sum exactly (e.g., sums with binomial distributions, or sums we can attack with generating functions), we don't need these. But they may be useful if computing the distribution of the sum directly is hard.

5.4.1 The Berry-Esseen theorem

The **Berry-Esseen theorem**[18] characterizes how quickly a sum of identical independent random variables converges to a normal distribution, as a function of the **third moment** of the random variables. Its simplest version says that if we have n independent, identically-distributed random variables $X_1 \dots X_n$, with $\mathrm{E}[X_i] = 0$, $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] = \sigma^2$, and $\mathrm{E}\left[|X_i|^3\right] \leq \rho$, then

$$\sup_{-\infty<x<\infty} \left| \Pr\left[\frac{1}{\sqrt{n}} \sum_{i=1}^{n} X_i \leq x\right] - \Phi(x) \right| \leq \frac{C\rho}{\sigma^3\sqrt{n}}, \tag{5.4.1}$$

were C is an absolute constant and Φ is the normal distribution function. Note that the σ^3 in the denominator is really $\mathrm{Var}[X_i]^{3/2}$. Since the probability bound doesn't depend on x, it's more useful toward the middle of the distribution than in the tails.

A classic proof of this result with $C = 3$ can be found in [Fel71, §XVI.5]. More recent work has tightened the bound on C: the best currently known constant is $C < 0.4784 < 1/2$, due to Shevtsova [She11].

As with most results involving sums of random variables, there are generalizations to martingales. These are too involved to describe here, but see [HH80, §3.6].

5.4.2 The Littlewood-Offord problem

The **Littlewood-Offord problem** asks, given a set of n complex numbers $x_1 \dots x_n$ with $|x_i| \geq 1$, for how many assignments of ± 1 to coefficients $\epsilon_1 \dots \epsilon_n$ it holds that $|\sum_{i=1}^{n} \epsilon_i x_i| \leq r$. Paul Erdős showed [Erd45] that this quantity was at most $cr2^n/\sqrt{n}$, where c is a constant. The quantity $c2^n/\sqrt{n}$ here is really $\frac{1}{2}\binom{n}{\lfloor n/2 \rfloor}$: Erdős's proof shows that for each interval of length $2r$, the number of assignments that give a sum in the interior of the interval is bounded by at most the sum of the r largest binomial coefficients.

In random-variable terms, this means that if we are looking at $\sum_{i=1}^{n} \epsilon x_i$, where the x_i are constants with $|x_i| \geq 1$ and the ϵ_i are independent ± 1 fair coin-flips, then $\Pr[|\sum_{i=1}^{n} \epsilon_i x_i| \leq r]$ is maximized by making all the x_i equal to 1. This shows that any distribution where the x_i are all reasonably large will not be any more concentrated than a binomial distribution.

There has been a lot of recent work on variants of the Littlewood-Offord problem, much of it by Terry Tao and Van Vu. See `http://terrytao.`

[18] Sometimes written *Berry-Esséen theorem* to help with the pronunciation of Esseen's last name.

wordpress.com/2009/02/16/a-sharp-inverse-littlewood-offord-theorem/
for a summary of much of this work.

Chapter 6

Randomized search trees

These are data structures that are either trees or equivalent to trees, and use randomization to maintain balance. We'll start by reviewing deterministic binary search trees and then add in the randomization.

6.1 Binary search trees

A **binary search tree** is a standard data structure for holding sorted data. A **binary tree** is either empty, or it consists of a **root** node containing a **key** and pointers to left and right **subtrees**. What makes a binary tree a binary search tree is the invariant that, both for the tree as a whole and any subtree, all keys in the left subtree are less than the key in the root, while all keys in the right subtree are greater than the key in the root. This ordering property means that we can search for a particular key by doing binary search: if the key is not at the root, we can recurse into the left or right subtree depending on whether it is smaller or bigger than the key at the root.

The efficiency of this operation depends on the tree being **balanced**. If each subtree always holds a constant fraction of the nodes in the tree, then each recursive step throws away a constant fraction of the remaining nodes. So after $O(\log n)$ steps, we find the key we are looking for (or find that the key is not in the tree). But the definition of a binary search tree does not by itself guarantee balance, and in the worst case a binary search tree degenerates into a linked list with $O(n)$ cost for all operations (see Figure 6.2.

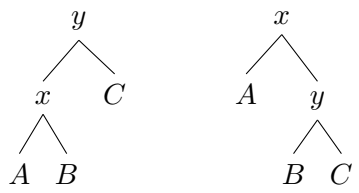


Figure 6.1: Tree rotations

6.1.1 Rebalancing and rotations

Deterministic binary search tree implementations include sophisticated rebalancing mechanisms to adjust the structure of the tree to preserve balance as nodes are inserted or delete. Typically this is done using **rotations**, which are operations that change the position of a parent and a child while preserving the left-to-right ordering of keys (see Figure 6.1).

Examples include **AVL trees** [AVL62], where the left and right subtrees of any node have heights that differ by at most 1; **red-black trees** [GS78], where a coloring scheme is used to maintain balance; and **scapegoat trees** [GR93], where no information is stored at a node but part of the tree is rebuilt from scratch whenever an operation takes too long. These all give $O(\log n)$ cost per operation (amortized in the case of scapegoat trees), and vary in how much work is needed in rebalancing. Both AVL trees and red-black trees perform more rotations than randomized rebalancing does on average.

6.2 Random insertions

Suppose we insert n keys into an initially-empty binary search tree in random order with no rebalancing. This means that for each insertion, we follow the same path that we would when searching for the key, and when we reach an empty tree, we replace it with a tree consisting solely of the key at the root.

Since we chose a random order, each element is equally likely to be the root, and all the elements less than the root end up in the left subtree, while all the elements greater than the root end up in the right subtree, where they are further partitioned recursively. This is exactly what happens in randomized QuickSort (see §1.3.1), so the structure of the tree will exactly mirror the structure of an execution of QuickSort. So, for example, we can immediately observe from our previous analysis of QuickSort that the **total path length**—the sum of the depths of the nodes—is $\Theta(n \log n)$, since the depth of each node is equal to 1 plus the number of comparisons it

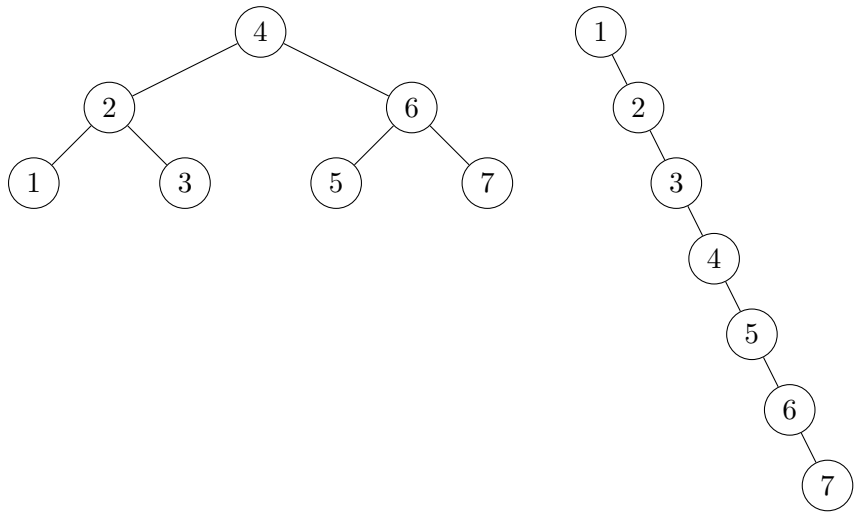


Figure 6.2: Balanced and unbalanced binary search trees

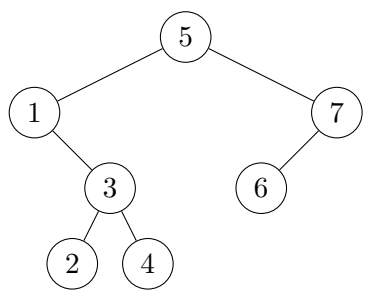


Figure 6.3: Binary search tree after inserting 5 1 7 3 4 6 2

participates in as a non-pivot, and (using the same argument as for Hoare's FIND in §3.6.3) that the height of the tree is $O(\log n)$ with high probability.[1]

When n is small, randomized binary search trees can look pretty scraggly. Figure 6.3 shows a typical example.

The problem with this approach in general is that we don't have any

[1]The argument for Hoare's FIND is that any node has at most 3/4 of the descendants of its parent on average; this gives for any node x that $\Pr[\text{depth}(x) > d] \leq (3/4)^{d-1}n$, or a probability of at most n^{-c} that $\text{depth}(x) > 1 + (c+1)\log(n)/\log(4/3) \approx 1 + 6.952 \ln n$ for $c = 1$, which we need to apply the union bound. The right answer for the actual height of a randomly-generated search tree in the limit is $4.31107 \ln n$ [Dev88] so this bound is actually pretty close. The real height is still nearly a factor of three worse than for a completely balanced tree, which has max depth bounded by $1 + \lg n \approx 1 + 1.44269 \ln n$.

guarantees that the input will be supplied in random order, and in the worst case we end up with a linked list, giving $O(n)$ worst-case cost for all operations.

6.3 Treaps

The solution to bad inputs is the same as for QuickSort: instead of assuming that the input is permuted randomly, we assign random priorities to each element and organize the tree so that elements with higher priorities rise to the top. The resulting structure is known as a **treap** [SA96], because it satisfies the binary search tree property with respect to keys and the **heap** property with respect to priorities.[2]

There's an extensive page of information on treaps at `http://faculty.washington.edu/aragon/treaps.html`, maintained by Cecilia Aragon, the co-inventor of treaps; they are also discussed at length in [MR95, §8.2]. We'll give a brief description here.

To insert a new node in a treap, first walk down the tree according to the key and insert the node as a new leaf. Then go back up fixing the heap property by rotating the new element up until it reaches an ancestor with the same or higher priority. (See Figure 6.4 for an example.) Deletion is the reverse of insertion: rotate a node down to a leaf (by swapping with its higher-priority child at each step), and then prune it off.

Because of the heap property, the root of each subtree is always the element in that subtree with the highest priority. This means that the structure of a treap is completely determined by the priorities and the keys, no matter what order the elements arrive in. We can imagine in retrospect that the treap is constructed recursively by choosing the highest-priority element as the root, then organizing all smaller-index and all larger-index nodes into the left and right subtrees by the same rule.

If we assign the priorities independently and uniformly at random from a sufficiently large set ($\omega(n^2)$ is enough in the limit), then we get no duplicates, and by symmetry all $n!$ orderings are equally likely. So the analysis of the depth of a treap with random priorities is identical to the analysis of a binary search tree with random insertion order. It's not hard to see that the costs

[2] The name "treap" for this data structure is now standard but the history is a little tricky. According to Seidel and Aragon, essentially the same data structure (though with non-random priorities) was previously called a cartesian tree by Vuillemin [Vui80], and the word "treap" was initially applied by McCreight to a different data structure—designed for storing two-dimensional data—that was called a "priority search tree" in its published form. [McC85].

```
1,60           1,60
   \              \
   2,3     -->    3,26
     \            /
     (3,26)     2,3


1,60             1,60             1,60            5,78
   \                \                \              /
   3,26      -->    3,26      -->   (5,78) -->   1,60
   /  \             /  \             /              \
 2,3  4,24        2,3  (5,78)      3,26           3,26
       \                /           /  \           /  \
       (5,78)         4,24       2,3  4,24      2,3  4,24

    5,78               5,78
   /    \             /    \
 1,60  6,18    -->  1,60    7,41
   \     \            \     /
  3,26  (7,41)       3,26  6,18
  /  \               /  \
2,3  4,24          2,3  4,24
```

Figure 6.4: Inserting values into a treap. Each node is labeled with k, p where k is the key and p the priority. Insertion of values not requiring rotations are not shown.

of search insertion, and deletion operations are all linear in the depth of the tree, so the expected cost of each of these operations is $O(\log n)$.

6.3.1 Assumption of an oblivious adversary

One caveat is that this only works if the priorities of the elements of the tree are in fact independent. If operations on the tree are chosen by an **adaptive adversary**, this assumption may not work. An adaptive adversary is one that can observe the choice made by the algorithm and react to them: in this case, a simple strategy would be to insert elements 1, 2, 3, 4, etc., in order, deleting each one and reinserting it until it has a lower priority value than all the smaller elements. This might take a while for the later elements, but the end result is the linked list again. For this reason it is standard to assume in randomized data structures that the adversary is **oblivious**, meaning that it has to specify a sequence of operations without knowing what choices are made by the algorithm. Under this assumption, whatever insert or delete operations the adversary chooses, at the end of any particular sequence of operations we still have independent priorities on all the remaining elements, and the $O(\log n)$ analysis goes through.

6.3.2 Analysis

The analysis of treaps as carried out by Seidel and Aragon [SA96] is a nice example of how to decompose a messy process into simple variables, much like the linear-of-expectation argument for QuickSort (§1.3.2). The key observation is that it's possible to bound both the expected depth of any node and the number of rotations needed for an insert or delete operation directly from information about the ancestor-descendant relationship between nodes.

Define two classes of indicator variables. For simplicity, we assume that the elements have keys 1 through n, which we also use as indices.

1. $A_{i,j}$ indicates the event that i is an **ancestor** of j, where i is an ancestor of j if it appears on the path from the root to j. Note that every node is an ancestor of itself.

2. $C_{i;\ell,m}$ indicates the event that i is a **common ancestor** of both ℓ and m; formally, $C_{i;\ell,m} = A_{i,\ell}A_{i,m}$.

The nice thing about these indicator variables is that it's easy to compute their expectations.

For $A_{i,j}$, i will be the ancestor of j if and only if i has a higher priority than j and there is no k between i and j that has an even higher priority: in other words, if i has the highest priority of all keys in the interval $[\min(i,j), \max(i,j)]$. To see this, imagine that we are constructing the treap recursively, by starting with all elements in a single interval and partitioning each interval by its highest-priority element. Consider the last interval in this process that contains both i and j, and suppose $i < j$ (the $j > i$ case is symmetric). If the highest-priority element is some k with $i < k < j$, then i and j are separated into distinct intervals and neither is the ancestor of the other. If the highest-priority element is j, then j becomes the ancestor of i. The highest-priority element can't be less than i or greater than j, because then we get a smaller interval that contains both i and j. So the only case where i becomes an ancestor of j is when i has the highest priority.

It follows that $\mathrm{E}[A_{i,j}] = \frac{1}{|i-j|+1}$, where the denominator is just the number of elements in the range $[\min(i,j), \max(i,j)]$.

For $C_{i;\ell,m}$, i is the common ancestor of both ℓ and m if and only if it is has the highest priority in both $[\min(i,\ell), \max(i,\ell)]$ and $[\min(i,m), \max(i,m)]$. It turns out that no matter what order i, ℓ, and m come in, these intervals overlap so that i must have the highest priority in $[\min(i,\ell,m), \max(i,\ell,m)]$. This gives $\mathrm{E}\left[C_[i;\ell,m]\right] = \frac{1}{\max(i,\ell,m)-\min(i,\ell,m)+1}$.

6.3.2.1 Searches

From the $A_{i,j}$ variables we can compute $\mathrm{depth}(j) = \sum_i A_{i,j} - 1$.[3] So

$$\begin{aligned}
\mathrm{E}[\mathrm{depth}(j)] &= \left(\sum_{i=1}^{n} \frac{1}{|i-j|+1}\right) - 1 \\
&= \left(\sum_{i=1}^{j} \frac{1}{j-i+1}\right) + \left(\sum_{i=j+1}^{n} \frac{1}{i-j+1}\right) - 1 \\
&= \left(\sum_{k=1}^{j} \frac{1}{k}\right) + \left(\sum_{k=2}^{n-j+1} \frac{1}{k}\right) - 1 \\
&= H_j + H_{n-j+1} - 2.
\end{aligned}$$

This is maximized at $j = (n+1)/2$, giving $2H_{(n+1)/2} - 2 = 2\ln n + O(1)$. So we get the same $2\ln n + O(1)$ bound on the expected depth of any one

[3] We need the -1 because of the convention that the root has depth 0, making the depth of a node one less than the number of its ancestors. Equivalently, we could exclude j from the sum and count only proper ancestors.

```
    4            2                6
   / \          / \              / \
 *2   6*   =>  1   4     or     4   7
 / \ / \          / \          / \
1 *3 5* 7       *3   6*      *2   5*
                    / \        \
                   5*  7       *3
```

Figure 6.5: Rotating 4 right shortens the right spine of its left subtree by removing 2; rotating left shortens the left spine of the right subtree by removing 6.

node that we got for QuickSort. We can also sum over all j to get the exact value of the expected total path length (but we won't). These quantities bound the expected cost of searches.

6.3.2.2 Insertions and deletions

For insertions and deletions, the question is how many rotations we have to perform to float a new leaf up to its proper location (after an insertion) or to float a deleted node down to a leaf (before a deletion). Since insertion is just the reverse of deletion, we can get a bound on both by concentrating on deletion. The trick is to find some metric for each node that (a) bounds the number of rotations needed to move a node to the bottom of the tree and (b) is easy to compute based on the A and C variables

The **left spine** of a subtree is the set of all nodes obtained by starting at the root and following left pointers; similarly the **right spine** is what we get if we follow the right pointers instead.

When we rotate an element down, we are rotating either its left or right child up. This removes one element from either the right spine of the left subtree or the left spine of the right subtree, but the rest of the spines are left intact (see Figure 6.5). Subsequent rotations will eventually remove all these elements by rotating them above the target, while other elements in the subtree will be carried out from under the target without ever appearing as a child or parent of the target. Because each rotation removes exactly one element from one or the other of the two spines, and we finish when both are empty, the sum of the length of the spines gives the number of rotations.

To calculate the length of the right spine of the left subtree of some element ℓ, start with the predecessor $\ell - 1$ of ℓ. Because there is no element

between them, either $\ell - 1$ is a descendant of ℓ or an ancestor of ℓ. In the former case (for example, when ℓ is 4 in Figure 6.5), we want to include all ancestors of $\ell - 1$ up to ℓ itself. Starting with $\sum_i A_{i,\ell-1}$ gets all the ancestors of $\ell - 1$, and subtracting off $\sum_i C_{i;\ell-1,\ell}$ removes any common ancestors of $\ell - 1$ and ℓ. Alternatively, if $\ell - 1$ is an ancestor of ℓ, every ancestor of $\ell - 1$ is also an ancestor of ℓ, so the same expression $\sum_i A_{i,\ell-1} - \sum_i C_{i;\ell-1,\ell}$ evaluates to zero.

It follows that the expected length of the right spine of the left subtree is exactly

$$\begin{aligned}
\mathrm{E}\left[\sum_{i=1}^{n} A_{i,\ell-1} - \sum_{i=1}^{n} C_{i;\ell-1,\ell}\right] &= \sum_{i=1}^{n} \frac{1}{|i-\ell|+1} - \sum_{i=1}^{n} \frac{1}{\max(i,\ell) - \min(i,\ell-1)+1} \\
&= \sum_{i=1}^{\ell} \frac{1}{\ell - i + 1} + \sum_{i=\ell+1}^{n} \frac{1}{i-\ell+1} - \sum_{i=1}^{\ell-1} \frac{1}{\ell-i+1} - \sum_{i=\ell}^{n} \frac{1}{i-(\ell-1)+1} \\
&= \sum_{j=1}^{\ell} \frac{1}{j} + \sum_{j=2}^{n-\ell+1} \frac{1}{j} - \sum_{j=2}^{\ell} \frac{1}{j} - \sum_{j=2}^{n-\ell+2} \frac{1}{j} \\
&= 1 - \frac{1}{n-\ell+1}.
\end{aligned}$$

By symmetry, the expected length of the left spine of the right subtree is $1 - \frac{1}{\ell}$. So the total expected number of rotations needed to delete the ℓ-th element is

$$2 - \frac{1}{\ell} + \frac{1}{n-\ell+1} \leq 2.$$

6.3.2.3 Other operations

Treaps support some other useful operations: for example, we can split a treap into two treaps consisting of all elements less than and all elements greater than a chosen pivot by rotating the pivot to the root ($O(\log n)$ rotations on average, equal to the pivot's expected depth as calculated in §6.3.2.1) and splitting off the left and right subtrees. The reverse merging operation has the same expected complexity.

6.4 Skip lists

A skip list [Pug90] is a randomized tree-like data structure based on linked lists. It consists of a level 0 list that is an ordinary sorted linked list, together

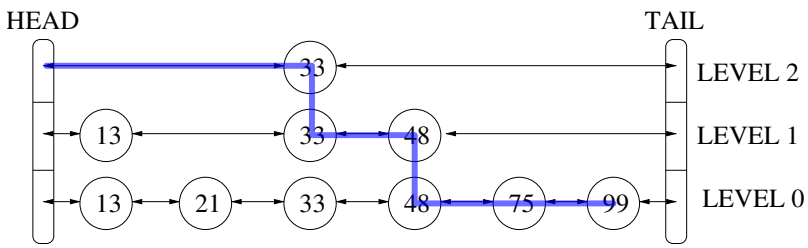


Figure 6.6: A skip list. The blue search path for 99 is superimposed on an original image from [AS07].

with higher-level lists that contain a random sampling of the elements at lower levels. When inserted into the level i list, an element flips a coin that tells it with probability p to insert itself in the level $i+1$ list as well. The result is that the element is represented by a tower of nodes, one in each of the bottom $1+X$ many layers, where X is a geometrically-distributed random variable. An example of a small skip list is shown in Figure 6.6.

Searches in a skip list are done by starting in the highest-level list and searching forward for the last node whose key is smaller than the target; the search then continues in the same way on the next level down. To bound the expected running time of a search, it helps to look at this process backwards; the reversed search path starts at level 0 and continues going backwards until it reaches the first element that is also in a higher level; it then jumps to the next level up and repeats the process. The nice thing about this reversed process is that it has a simple recursive structure: if we restrict a skip list to only those nodes to the left of and at the same level or higher of a particular node, we again get a skip list. Furthermore, the structure of this restricted skip list depends only on coin-flips taken at nodes within it, so it's independent of anything that happens elsewhere in the full skip list.

We can analyze this process by tracking the number of nodes in the restricted skip list described above, which is just the number of nodes in the current level that are earlier than the current node. If we move left, this drops by 1; if up, this drops to p times its previous value on average. So the number of such nodes X_k after k steps satisfies $\mathrm{E}[X_{k+1} \mid X_k] = (1-p)(X_k - 1) + p(p\,\mathrm{E}[X_k]) = (1-p+p^2)X_k - (1-p)^2 \leq (1-p+p^2)X_k$, and in general we get $\mathrm{E}[X_k] \leq (1-p+p^2)^k X_0 = (1-p+p^2)n$. (We can substitute the rank of the starting node for n if we want a slightly tighter bound.) This is minimized at $p = 1/2$, giving $\mathrm{E}[X_k] \leq (3/4)^k n$, suspiciously similar to the bound we computed before for random binary search trees.

When $X_k = 0$, our search is done, so if T is the time to search, we have $\Pr[T \geq k] = \Pr[X_k \geq 1] \leq (3/4)^k n$, by Markov's inequality. In particular, if we want to guarantee that we finish with probability $1 - \epsilon$, we need to run for $\log_{4/3}(n/\epsilon)$ steps. This translates into an $O(\log n)$ bound on the search time, and the constant is even the same as our (somewhat loose) bound for treaps.

The space per element of a skip list also depends on p. Every element needs one pointer for each level it appears in. This gives one pointer for level 0, plus $1/(1-p)$ pointers on average for the higher levels. For constant p this is $O(1)$. However, the space cost can reduced (at the cost of increasing search time) by adjusting p. For example, if space is at a premium, setting $p = 1/10$ produces $10/9$ pointers per node on average—not much more than in a linked list—but still gives $O(\log n)$ search time. In general the trade-off is between $n\left(1 + \frac{1}{1-p}\right)$ total expected space and $\log_{1/(1-p+p^2)}(n/\epsilon)$ search time. The constant factor in the search time is $\frac{1}{-\log(1-p+p^2)}$, which for small p is approximately $1/p$.

Like treaps, skip lists can be split an merged in a straightforward way. The difference is that in a skip list, it's enough to cut (or recreate) all the pointers crossing the boundary, without changing the structure of the rest of the list.

Chapter 7

Hashing

The basic idea of hashing is that we have keys from a large set U, and we'd like to pack them in a small set M by passing them through some function $h : U \to M$, without getting too many **collisions**, pairs of distinct keys x and y with $h(x) = h(y)$. Where randomization comes in is that we want this to be true even if the adversary picks the keys to hash. At one extreme, we could use a random function, but this will take a lot of space to store.[1] So our goal will be to find functions with succinct descriptions that are still random enough to do what we want.

The presentation in this chapter is based largely on [MR95, §§8.4-8.5] (which is in turn based on work of Carter and Wegman [CW77] on universal hashing and Fredman, Komlós, and Szemerédi [FKS84] on $O(1)$ worst-case hashing); on [PR04] and [Pag06] for cuckoo hashing; and [MU05, §5.5.3] for Bloom filters.

7.1 Hash tables

Here we review the basic idea of **hash tables**, which are implementations of the **dictionary** data type mapping keys to values. The basic idea of hash tables is usually attributed to Dumey [Dum56].[2]

[1] An easy counting argument shows that almost all functions from U to M take $|U| \log |M|$ bits to represent, no matter how clever you are about choosing your representation. This forms the basis for **algorithmic information theory**, which *defines* an object as random if there is no way to reduce the number of bits used to express it.

[2] Caveat: This article is pretty hard to find, so I am basing this citation on its frequent appearance in later sources. This is generally a bad idea that would not really be acceptable in an actual scholarly publication.

U	Universe of all keys
$S \subseteq U$	Set of keys stored in the table
$n = \lvert S \rvert$	Number of keys stored in the table
M	Set of table positions
$m = \lvert M \rvert$	Number of table positions
$\alpha = n/m$	Load factor

Table 7.1: Hash table parameters

Suppose we want to store n elements from a universe U of in a table with **keys** or **indices** drawn from an index space M of size m. Typically we assume $U = [|U|] = \{0 \ldots |U| - 1\}$ and $M = [m] = \{0 \ldots m - 1\}$.

If $|U| \leq m$, we can just use an array. Otherwise, we can map keys to positions in the array using a **hash function** $h : U \to M$. This necessarily produces **collisions**: pairs (x, y) with $h(x) = h(y)$, and any design of a hash table must include some mechanism for handling keys that hash to the same place. Typically this is a secondary data structure in each bin, but we may also place excess values in some other place. Typical choices for data structures are linked lists (**separate chaining** or just **chaining**) or secondary hash tables (see §7.3 below). Alternatively, we can push excess values into other positions in the same hash table (**open addressing** or **probing**) or another hash table (see §7.4).

For all of these techniques, the cost will depend on how likely it is that we get collisions. An adversary that knows our hash function can always choose keys with the same hash value, but we can avoid that by choosing our hash function randomly. Our ultimate goal is to do each search in $O(1 + n/m)$ expected time, which for $n \leq m$ will be much better than the $\Theta(\log n)$ time for pointer-based data structures like balanced trees or skip lists. The quantity n/m is called the **load factor** of the hash table and is often abbreviated as α.

All of this only works if we are working in a RAM (random-access machine model), where we can access arbitrary memory locations in time $O(1)$ and similarly compute arithmetic operations on $O(\log|U|)$-bit values in time $O(1)$. There is an argument that in reality any actual RAM machine requires either $\Omega(\log m)$ time to read one of m memory locations (routing costs) or, if one is particularly pedantic, $\Omega(m^{1/3})$ time (speed of light + finite volume for each location). We will ignore this argument.

We will try to be consistent in our use of variables to refer to the different parameters of a hash table. Table 7.1 summarizes the meaning of these variable names.

7.2 Universal hash families

A family of hash functions H is **2-universal** if for any $x \neq y$, $\Pr[h(x) = h(y)] \leq 1/m$ for a uniform random $h \in H$. It's **strongly 2-universal** if for any $x_1 \neq x_2 \in U$, $y_1, y_2 \in M$, $\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = 1/m^2$ for a uniform random $h \in H$. Another way to describe strong 2-universality is that the values of the hash function are uniformly distributed and pairwise-independent.

For $k > 2$, **k-universal** usually means **strongly k-universal**: Given distinct $x_1 \ldots x_k$, and any $y_1 \ldots y_k$, $\Pr[h(x_i) = y_i) \forall i] = m^{-k}$. This is equivalent to uniform distribution and k-wise independence. It is possible to generalize the weak version of 2-universality to get a weak version of k-universality ($\Pr[h(x_i) \text{ are all equal}] \leq m^{-(k-1)}$), but this generalization is not as useful as strong k-universality.

To analyze universal hash families, it is helpful to have some notation for counting collisions. We'll mostly be doing counting rather than probabilities because it saves carrying around a lot of denominators. Since we are assuming uniform choices of h we can alway get back probabilities by dividing by $|H|$.

Let $\delta(x, y, h) = 1$ if $x \neq y$ and $h(x) = h(y)$, 0 otherwise. Abusing notation, we also define, for sets X, Y, and H, $\delta(X, Y, H) = \sum_{x \in X, y \in Y, h \in H} \delta(x, y, h)$, with e.g. $\delta(x, Y, h) = \delta(\{x\}, Y, \{h\})$. Now the statement that H is 2-universal becomes $\forall x, y : \delta(x, y, H) \leq |H|/m$; this says that only a fraction of $1/m$ of the functions in H cause any particular distinct x and y to collide.

If H includes all functions $U \to M$, we get equality: a random function gives $h(x) = h(y)$ with probability exactly $1/m$. But we might do better if each h tends to map distinct values to distinct places. The following lemma shows we can't do too much better:

Lemma 7.2.1. *For any family H, there exist x, y such that $\delta(x, y, H) \geq \frac{|H|}{m}\left(1 - \frac{m-1}{|U|-1}\right)$.*

Proof. We'll count collisions in the inverse image of each element z. Since all distinct pairs of elements of $h^{-1}(z)$ collide with each other, we have

$$\delta(h^{-1}(z), h^{-1}(z), h) = \left|h^{-1}(z)\right| \cdot \left(\left|h^{-1}(z)\right| - 1\right).$$

Summing over all $z \in M$ gets all collisions, giving

$$\delta(U, U, h) = \sum_{z \in M} \left(\left|h^{-1}(z)\right| \cdot \left(\left|h^{-1}(z)\right| - 1\right)\right).$$

Use convexity or Lagrange multipliers to argue that the right-hand side is minimized subject to $\sum_z |h^{-1}(z)| = |U|$ when all pre-images are the same size $|U|/m$. It follows that

$$\begin{aligned}\delta(U,U,h) &\geq \sum_{z\in M} \frac{|U|}{m}\left(\frac{|U|}{m}-1\right)\\ &= m\frac{|U|}{m}\left(\frac{|U|}{m}-1\right)\\ &= \frac{|U|}{m}(|U|-m).\end{aligned}$$

If we now sum over all h, we get

$$\delta(U,U,H) \geq \frac{|H|}{m}|U|(|U|-m).$$

There are exactly $|U|(|U|-1)$ ordered pairs x, y for which $\delta(x,y,H)$ might not be zero; so the Pigeonhole principle says some pair x, y has

$$\begin{aligned}\delta(x,y,H) &\geq \frac{|H|}{m}\left(\frac{|U|(|U|-m)}{|U|(|U|-1)}\right)\\ &= \frac{|H|}{m}\left(1-\frac{m-1}{|U|-1}\right).\end{aligned}$$

□

Since $1-\frac{m-1}{|U|-1}$ is likely to be very close to 1, we are happy if we get the 2-universal upper bound of $|H|/m$.

Why we care about this: With a 2-universal hash family, chaining using linked lists costs $O(1+s/n)$ expected time per operation. The reason is that the expected cost of an operation on some key x is proportional to the size of the linked list at $h(x)$ (plus $O(1)$ for the cost of hashing itself). But the expected size of this linked list is just the expected number of keys y in the dictionary that collide with x, which is exactly $s\delta(x,y,H) \leq s/n$.

7.2.1 Linear congruential hashing

Universal hash families often look suspiciously like classic pseudorandom number generators. Here is a 2-universal hash family based on taking remainders. It is assumed that the universe U is a subset of $\mathbb{Z}_p$, the integers mod p; effectively, this just means that every element x of U satisfies $0 \leq x \leq p-1$.

Lemma 7.2.2. *Let* $h_{ab}(x) = (ax+b \bmod p) \bmod m$*, where* $a \in \mathbb{Z}_p - \{0\}$, $b \in \mathbb{Z}_p$*, and* p *is a prime* $\geq m$*. Then* $\{h_{ab}\}$ *is* 2*-universal.*

Proof. Again, we count collisions. Split $h_{ab}(x)$ as $g(f_{ab}(x))$ where $f_{ab}(x) = ax + b \bmod p$ and $g(x) = x \bmod m$.

The intuition is that after feeding any distinct x and y through f_{ab}, all distinct pairs of values r and s are equally likely. We then show that feeding these values to g produces no more collisions than expected.

The formal statement of the intuition is that for any $0 \leq x, y \leq p-1$ with $x \neq y$, $\delta(x, y, H) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

To prove this, fix x and y, and consider some pair $r \neq s \in Z_p$. Then the equations $ax+b = t$ and $ay+b = u$ have a unique solution for a and b mod p (because $\mathbb{Z}_p$ is a finite field). So $(f_{ab}(x), f_{ab}(y))$ is a bijection between pairs a, b and pairs u, u and thus $\delta(x, y, H) = \sum_{a,b} \delta(x, y, f_{ab}) = \sum_{t \neq u} \delta(t, u, g) = \delta(\mathbb{Z}_p, \mathbb{Z}_p, g)$.

Now we just need to count how many distinct t and u collide. There are p choices for t. For each t, there are at most $\lceil p/m \rceil$ values that map to the same remainder mod m, but one of those values is equal to t, so this leaves only $\lceil p/m \rceil - 1$ choices for u. But $\lceil p/m \rceil - 1 \leq (p+(m-1))/m - 1 = (p-1)/m$. So $\delta(\mathbb{Z}_p, \mathbb{Z}_p, g) \leq p(p-1)/m$.

Since each choice of t and u occurs with probability $\frac{1}{p(p-1)}$, this gives a probability of a collision of at most $1/m$. □

A difficulty with this hash family is that it requires doing modular arithmetic. A faster hash is given by Dietzfelbinger *et al.* [DHKP97], although it requires a slight weakening of the notion of 2-universality. For each k and ℓ they define a class $H_{k,\ell}$ of functions from $[2^k]$ to $[2^\ell]$ by defining

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell},$$

where $x \operatorname{div} y = \lfloor x/y \rfloor$. They prove [DHKP97, Lemma 2.4] that if a is a random *odd* integer with $0 < a < 2^\ell$, and $x \neq y$, $\Pr[h_a(x) = h_a(y)] \leq 2^{-\ell+1}$. This increases by a factor of 2 the likelihood of a collision, but any extra costs from this can often be justified in practice by the reduction in costs from working with powers of 2.

If we are willing to use more randomness (and more space), a method called **tabulation hashing** (§7.2.2) gives a simpler alternative that is 3-universal.

7.2.2 Tabulation hashing

Tabulation hashing [CW77] is a method for hashing fixed-length strings (or things that can be represented as fixed-length strings) into bit-vectors. The description here follows Patrascu and Thorup [PT12].

Let c be the length of each string in characters, and let s be the size of the alphabet. Initialize the hash function by constructing tables $T_1 \dots T_c$ mapping characters to independent random bit-vectors of size $\lg m$. Define

$$h(x) = T_1[x_1] \oplus T_2[x_2] \oplus \dots T_c[x_c],$$

where $\oplus$ represents bitwise exclusive OR (what ^ does in C-like languages).[3] This gives a family of hash functions that is 3-wise independent but not 4-wise independent.

To show independence, the intuition is that if we can order the strings $x^1, x^2, \dots, x^n$ that we are hashing so that each has a position i_j such that $x^j_{i_j} \neq x^{j'}_{i_j}$ for any $j' < j$, then we have, for each value v, $\Pr\left[h(x^j) = v \;\middle|\; h(x^{j'}) = v_{j'}, \forall j' < j\right] = 1/m$. It follows that the hash values are independent:

$$\begin{aligned}
\Pr\left[h(x^1) = v_1, h(x^2) = v_2, \dots, h(x^n) = v_n)\right] &= \prod_{j=1}^{n} \Pr\left[h(x^j) = v_j \;\middle|\; h(x^1) = v_1 \dots h(x^{j-1}) = v_{j-1}\right] \\
&= \frac{1}{m^n} \\
&= \prod_{j=1}^{n} \Pr\left[h(x^j) = v_j\right].
\end{aligned}$$

Now we want to show that when $n = 3$, this actually works for all possible distinct strings x, y, and z. Let S be the set of indices i such that $y_i \neq x_i$, and similarly let T be the set of indices i such that $z_i \neq x_i$; note that both sets must be non-empty, since $y \neq x$ and $z \neq x$. If $S \setminus T$ is nonempty, then (a) there is some index i in T where $z_i \neq x_i$, and (b) there is some index j in $S \setminus T$ where $y_i \neq x_i = z_i$; in this case, ordering the strings as x, z, y gives the independence property above. If $T \setminus S$ is nonempty, order them as x, z, y instead. Alternatively, if $S = T$, then $y_i \neq z_i$ for some i in S (otherwise $y = z$, since they both equal x on all positions outside S). In this case, x_i, y_i, and z_i are all distinct.

For $n = 4$, we can have strings `aa`, `ab`, `ba`, and `bb`. If we take the bitwise exclusive OR of all four hash values, we get zero, because each character is

[3] Letting m be a power of 2 and using exclusive OR is convenient on real computers. If for some reason we don't like this approach, the same technique, with essentially the same analysis, works for arbitrary m if we replace XOR with addition mod m.

included exactly twice in each position. So the hash values are not independent, and we do not get 4-independence in general.

However, even though tabulation hashing is not 4-independent, most reasonably small sets of inputs do give independence. This can be used to show various miraculous properties like working well for cuckoo hashing.

7.3 FKS hashing

The FKS hash table, named for Fredman, Komlós, and Szemerédi [FKS84], is a method for storing a static set S so that we never pay more than constant time for search (not just in expectation), while at the same time not consuming too much space. The assumption that S is static is critical, because FKS chooses hash functions based on the elements of S.

If we were lucky in our choice of S, we might be able to do this with standard hashing. A **perfect hash function** for a set $S \subseteq U$ is a hash function $h : U \to M$ that is injective on S (that is, $x \neq y \to h(x) \neq h(y)$ when $x, y \in S$). Unfortunately, we can only count on finding a perfect hash function if m is large:

Lemma 7.3.1. *If H is 2-universal and $|S| = n$ with $n^2 \leq m$, then there is a perfect $h \in H$ for S.*

Proof. We'll do the usual collision-counting argument. For all $x \neq y$, we have $\delta(x, y, H) \leq |H|/m$. So $\delta(S, S, H) \leq n(n-1)|H|/m$. The Pigeonhole Principle says that there exists a particular $h \in H$ with $\delta(S, S, h) \leq n(n-1)/m < n^2/m \leq 1$. But $\delta(S, S, h)$ is an integer, so it can only be less than 1 by being equal to 0: no collisions. □

Essentially the same argument shows that if $n^2 \leq \alpha m$, then $\Pr[h \text{ is perfect for } S] \geq 1 - \alpha$. This can be handy if we want to find a perfect hash function and not just demonstrate that it exists.

Using a perfect hash function, we get $O(1)$ search time using $O(n^2)$ space. But we can do better by using perfect hash functions only at the second level of our data structure, which at top level will just be an ordinary hash table. This is the idea behind the Fredman-Komlós-Szemerédi (FKS) hash table [FKS84].

The short version is that we hash to $n = |S|$ bins, then rehash perfectly within each bin. The top-level hash table stores a pointer to a header for each bin, which gives the size of the bin and the hash function used within it. The i-th bin, containing n_i elements, uses $O(n_i^2)$ space to allow perfect

hashing. The total size is $O(n)$ as long as we can show that $\sum_{i=1}^{n} n_i^2 = O(n)$. The time to do a search is $O(1)$ in the worst case: $O(1)$ for the outer hash plus $O(1)$ for the inner hash.

Theorem 7.3.2. *The FKS hash table uses $O(n)$ space.*

Proof. Suppose we choose $h \in H$ as the outer hash function, where H is some 2-universal family of hash functions. Compute:

$$\begin{aligned}\sum_{i=1}^{n} n_i^2 &= \sum_{i=1}^{n}(n_i + n_i(n_i - 1)) \\ &= n + \delta(S, S, h).\end{aligned}$$

Since H is 2-universal, we have $\delta(S, S, H) \leq |H|s(s-1)/n$. But then the Pigeonhole principle says there exists some $h \in H$ with $\delta(S, S, h) \leq \frac{1}{|H|}\delta(S, S, H) \leq n(n-1)/n = n - 1$. This gives $\sum_{i=1}^{n} n_i^2 \leq n + (n-1) = 2n - 1 = O(n)$. □

If we want to find a good h quickly, increasing the size of the outer table to n/α gives us a probability of at least $1 - \alpha$ of getting a good one, using essentially the same argument as for perfect hash functions.

7.4 Cuckoo hashing

Goal: Get $O(1)$ search time in a dynamic hash table at the cost of a messy insertion procedure. In fact, each search takes only two reads, which can be done in parallel; this is optimal by a lower bound of Pagh [Pag01], which shows a matching upper bound for static dictionaries. **Cuckoo hashing** is an improved version of this result that allows for dynamic insertions.

Cuckoo hashing was invented by Pagh and Rodler [PR04]; the version described here is based on a simplified version from notes of Pagh [Pag06] (the main difference is that it uses just one table instead of the two tables—one for each hash function—in [PR04]).

7.4.1 Structure

We have a table T of size n, with two separate, independent hash functions h_1 and h_2. These functions are assumed to be k-universal for some sufficiently large value k; as long as we never look at more than k values at once, this means we can treat them effectively as random functions. In practice, using crummy hash functions seems to work just fine, a common property of hash

tables. There are also specific hash functions that have been shown to work with particular variants of cuckoo hashing [PR04, PT12]. We will avoid these issues by assuming that our hash functions are actually random.

Every key x is stored either in $T[h_1(x)]$ or $T[h_2(x)]$. So the search procedure just looks at both of these locations and returns whichever one contains x (or fails if neither contains x).

To insert a value $x_1 = x$, we must put it in $T[h_1(x_1)]$ or $T[h_2(x_1)]$. If one or both of these locations is empty, we put it there. Otherwise we have to kick out some value that is in the way (this is the "cuckoo" part of cuckoo hashing, named after the bird that leaves its eggs in other birds' nests). We do this by letting $x_2 = T[h_1(x_1)]$ and writing x_1 to $T[h_1(x_1)]$. We now have a new "nestless" value x_2, which we swap with whatever is in $T[h_2(x_2)]$. If that location was empty, we are done; otherwise, we get a new value x_3 that we have to put in $T[h_1(x_3)]$ and so on. The procedure terminates when we find an empty spot or if enough iterations have passed that we don't expect to find an empty spot, in which case we rehash the entire table. This process can be implemented succinctly as shown in Algorithm 7.1.

```
procedure insert(x)
    if T(h_1(x) = x or T(h_2(x)) = x then
        return
    pos ← h_1(x)
    for i ← 1 ... n do
        if T[pos] = ⊥ then
            T[pos] ← x
            return
        x ⇄ T[pos]
        if pos = h_1(x) then
            pos ← h_2(x)
        else
            pos ← h_1(x)
    If we got here, rehash the table and reinsert x.
```

Algorithm 7.1: Insertion procedure for cuckoo hashing. Adapted from [Pag06]

A detail not included in the above code is that we always rehash (in theory) after m^2 insertions; this avoids potential problems with the hash functions used in the paper not being universal enough. We will avoid this

issue by assuming that our hash functions are actually random (instead of being approximately n-universal with reasonably high probability). For a more principled analysis of where the hash functions come from, see [PR04]. An alternative hash family that is known to work for a slightly different variant of cuckoo hashing is tabulation hashing, as described in §7.2.2; the proof that this works is found in [PT12].

7.4.2 Analysis

The main question is how long it takes the insertion procedure to terminate, assuming the table is not too full.

First let's look at what happens during an insert if we have a lot of nestless values. We have a sequence of values $x_1, x_2, \ldots$, where each pair of values x_i, x_{i+1} collides in h_1 or h_2. Assuming we don't reach the loop limit, there are three main possibilities (the leaves of the tree of cases below):

1. Eventually we reach an empty position without seeing the same key twice.

2. Eventually we see the same key twice; there is some i and $j > i$ such that $x_j = x_i$. Since x_i was already moved once, when we reach it the second time we will try to move it back, displacing x_{i-1}. This process continues until we have restored x_2 to $T[h_1(x_1)]$, displacing x_1 to $T[h_2(x_1)]$ and possibly creating a new sequence of nestless values. Two outcomes are now possible:

   (a) Some x_ℓ is moved to an empty location. We win!

   (b) Some x_ℓ is moved to a location we've already looked at. We lose! We find we are playing musical chairs with more players than chairs, and have to rehash.

Let's look at the probability that we get the last, *closed loop* case. Following Pagh-Rodler, we let v be the number of distinct nestless keys in the loop. Since v includes x_1, v is at least 1. We can now count how many different ways such a loop can form.

There are at most v^3 choices for i, j, and ℓ, m^{v-1} choices of cells for the loop, and n^{v-1} choices for the non-x_1 elements of the loop. For each non-x_i element, its initial placement may determined by either h_1 or h_2; this gives another 2^{v-1} choices.[4] This gives a total of $v^3(2nm)^{v-1}$ possible closed loops starting with x_1 that have v distinct nodes.

[4] The original analysis in [PR04] avoids this by alternating between two tables, so that we can determine which of h_1 or h_2 is used at each step by parity.

Since each particular loop allows us to determine both h_1 and h_2 for all v of its elements, the probability that we get exactly these hash values (so that the loop occurs) is m^{-2v}. Summing over all closed loops with v elements gives a total probability of $v^2(2nm)^{v-1}m^{-2v} = v^3(2n/m)^{v-1}m^{-v-1} \leq v^3(2n/m)^{v-1}m^{-2}$.

Now sum over all $v \geq 1$. We get $m^{-2}\sum_{v=1}^{n} v^3(2n/m)^{v-1} < m^{-2}\sum_{v=1}^{\infty} v^3(2n/m)^{v-1}$. The series converges if $2n/m < 1$, so for any fixed $\alpha < 1/2$, the probability of any closed loop forming is $O(m^{-2}$. Since the cost of hitting a closed loop is $O(n+m) = O(m)$, this adds only $O(m^{-1})$ to the insertion complexity.

Now we look at what happens if we don't get a closed loop. This doesn't force us to rehash, but if the path is long enough, we may still pay a lot to do an insertion.

It's a little messy to analyze the behavior of keys that appear more than once in the sequence, so the trick used in the paper is to observe that for any sequence of nestless keys $x_1 \ldots x_p$, there is a subsequence of size $p/3$ with no repetitions that starts with x_1. This will be either the sequence S_1 given by $x_1 \ldots x_{j-1}$—the sequence starting with the first place we try to insert x_1–or S_2 given by $x_1 = x_{i+j-1} \ldots x_p$, the sequence starting with the second place we try to insert x_1. Between these we have a third sequence S_3 where we undo some of the moves made in S_1. Because $|S_1| + |S_3| + |S_2| \geq p$, at least one of thee subsequences has size $p/3$. But $|S_3| \leq |S_1|$, so it must be either S_1 or S_2.

We can then argue that the probability that we get a sequence of v distinct keys in either S_1 or S_2 most $2(n/m)^{v-1}$ (since we have to hit a nonempty spot, with probability at most n/m, at each step, but there are two possible starting locations), which gives an expected insertion time bounded by $\sum 3v(n/m)^{v-1} = O(1)$, assuming n/m is bounded by a constant less than 1. Since we already need $n/m \leq 1/2$ to avoid the bad closed-loop case, we can use this here as well.

An annoyance with cuckoo hashing is that it has high space overhead compared to more traditional hash tables: in order for the first part of the analysis above to work, the table must be at least half empty. This can be avoided at the cost of increasing the time complexity by choosing between d locations instead of 2. This technique, due to Fotakis *et al.* [FPSS03], is known as **d-ary cuckoo hashing**; for suitable choice of d it uses $(1+\epsilon)n$ space and guarantees that a lookup takes $O(1/\epsilon)$ probes while insertion takes $(1/\epsilon)^{O(\log\log(1/\epsilon))}$ steps in theory and appears to take $O(1/\epsilon)$ steps in experiments done by the authors.

7.5 Practical issues

For large hash tables, local probing schemes are faster, because it is likely that all of the locations probed to find a particular value will be on the same virtual memory page. This means that a search for a new value usually requires one cache miss instead of two. **Hopscotch hashing** [HST08] combines ideas from linear probing and cuckoo hashing to get better performance than both in practice.

Hash tables that depend on strong properties of the hash function may behave badly if the user supplies a crummy hash function. For this reason, many library implementations of hash tables are written defensively, using algorithms that respond better in bad cases. See `http://svn.python.org/view/python/trunk/Objects/dictobject.c` for an example of a widely-used hash table implementation chosen specifically because of its poor theoretical characteristics.

7.6 Bloom filters

See [MU05, §5.5.3] for basics and a formal analysis or `http://en.wikipedia.org/wiki/Bloom_filter` for many variations and the collective wisdom of the unwashed masses. The presentation here mostly follows [MU05].

7.6.1 Construction

Bloom filters are a highly space-efficient randomized data structure invented by Burton H. Bloom [Blo70] that store sets of data, with a small probability that elements not in the set will be erroneously reported as being in the set.

Suppose we have k independent hash functions $h_1, h_2, \ldots, h_k$. Our memory store A is a vector of m bits, all initially zero. To store a key x, set $A[h_i(x)] = 1$ for all i. To test membership for x, see if $A[h_i(x)] = 1$ for all i. The membership test always gives the right answer if x is in fact in the Bloom filter. If not, we might decide that x is in the Bloom filter anyway.

7.6.2 False positives

The probability of such **false positives** can be computed in two steps: first, we estimate how many of the bits in the Bloom filter are set after inserting n values, and then we use this estimate to compute a probability that any fixed x shows up when it shouldn't.

If the h_i are close to being independent random functions,[5] then with n entries in the filter we have $\Pr[A[i]=1] = 1-(1-1/m)^{kn}$, since each of the kn bits that we set while inserting the n values has one chance in m of hitting position i.

We'd like to simplify this using the inequality $1+x \le e^x$, but it goes in the wrong direction; instead, we'll use $1-x \ge e^{-x-x^2}$, which holds for $0 \le x \le 0.683803$ and in our application holds for $m \ge 2$. This gives

$$\begin{aligned}\Pr[A[i]=1] &\le 1-(1-1/m)^{kn}\\ &\le 1-e^{-k(n/m)(1+1/m)}\\ &= 1-e^{-k\alpha(1+1/m)}\\ &= 1-e^{-k\alpha'}\end{aligned}$$

where $\alpha = n/m$ is the load factor and $\alpha' = \alpha(1+1/m)$ is the load factor fudged upward by a factor of $1+1/m$ to make the inequality work.

Suppose now that we check to see if some value x that we never inserted in the Bloom filter appears to be present anyway. This occurs if $A[h_i(x)]=1$ for all i. Since each $h_i(x)$ is assumed to be an independent uniform probe of A, the probability that they all come up 1 conditioned on A is

$$\left(\frac{\sum A[i]}{m}\right)^k. \tag{7.6.1}$$

We have an upper bound $\mathrm{E}\left[\sum A[i]\right] \le m\left(1-e^{-k\alpha'}\right)$, and if we were born luckier might be able to get an upper bound on the expectation of (7.6.1) by applying Jensen's inequality to the function $f(x)=x^k$. But sadly this inequality also goes in the wrong direction, because f is convex for $k>1$. So instead we will prove a concentration bound on $S=\sum A[i]$.

Because the $A[i]$ are not independent, we can't use off-the-shelf Chernoff bounds. Instead, we rely on McDiarmid's inequality. Our assumption is that the locations of the kn ones that get written to A are independent. Furthermore, changing the location of one of these writes changes S by at most 1. So McDiarmid's inequality (5.3.11) gives $\Pr[S \ge \mathrm{E}[S]+t] \le e^{-2t^2/kn}$, which is bounded by n^{-c} for $t \ge \sqrt{\frac{1}{2}ckn\log n}$. So as long as a reasonably large fraction of the array is likely to be full, the relative error

[5] We are going sidestep the rather deep swamp of how plausible this assumption is and what assumption we should be making instead; however, it is known [KM08] that starting with two sufficiently random-looking hash functions h and h' and setting $h_i(x) = h(x)+ih'(x)$ works.

from assuming $S = \mathrm{E}[S]$ is likely to be small. Alternatively, if the array is mostly empty, then we don't care about the relative error so much because the probability of getting a false positive will already be exponentially small as a function of k.

So let's assume for simplicity that our false positive probability is exactly $(1 - e^{-k\alpha'})^k$. We can choose k to minimize this quantity for fixed α' by doing the usual trick of taking a derivative and setting it to zero; to avoid weirdness with the k in the exponent, it helps to take the logarithm first (which doesn't affect the location of the minimum), and it further helps to take the derivative with respect to $x = e^{-\alpha' k}$ instead of k itself. Note that when we do this, $k = -\frac{1}{\alpha'} \ln x$ still depends on x, and we will deal with this by applying this substitution at an appropriate point.

Compute

$$\begin{aligned}
\frac{d}{dx} \ln\left((1-x)^k\right) &= \frac{d}{dx} k \ln(1-x) \\
&= \frac{d}{dx}\left(-\frac{1}{\alpha'} \ln x\right) \ln(1-x) \\
&= -\frac{1}{\alpha'}\left(\frac{\ln(1-x)}{x} - \frac{\ln x}{1-x}\right).
\end{aligned}$$

Setting this to zero gives $(1-x)\ln(1-x) = x \ln x$, which by symmetry has the unique solution $x = 1/2$, giving $k = \frac{1}{\alpha'} \ln 2$.

In other words, to minimize the false positive rate for a known load factor α, we want to choose $k = \frac{1}{\alpha'} \ln 2 = \frac{1}{\alpha(1+1/m)} \ln 2$, which makes each bit one with probability approximately $1 - e^{-\ln 2} = \frac{1}{2}$. This makes intuitive sense, since having each bit be one or zero with equal probability maximizes the entropy of the data.

The probability of a false positive is then $2^{-k} = 2^{-\ln 2/\alpha'}$. For a given maximum false positive rate ϵ, and assuming optimal choice of k, we need to keep $\alpha' \leq \frac{\ln^2 2}{\ln(1/\epsilon)}$ or $\alpha \leq \frac{\ln^2 2}{(1+1/m)\ln(1/\epsilon)}$.

Alternatively, if we fix ϵ and n, we need $m/(1+1/m) \geq n \cdot \frac{\ln(1/\epsilon)}{\ln^2 2} \approx 1.442 n \lg(1/\epsilon)$, which works out to $m \geq 1.442 n \lg(1/\epsilon) + O(1)$. This is very good for constant ϵ.

Note that for this choice of m, we have $\alpha = O(1/\ln(1/\epsilon))$, giving $k = O(\log\log(1/\epsilon))$. So for polynomial ϵ, we get $k = O(\log\log n)$. This means that not only do we use little space, but we also have very fast lookups (although not as fast as the $O(1)$ cost of a real hash table).

7.6.3 Comparison to optimal space

If we wanted to design a Bloom-filter-like data structure from scratch and had no constraints on processing power, we'd be looking for something that stored an index of size $\lg M$ into a family of subsets $S_1, S_2, \ldots S_M$ of our universe of keys U, where $|S_i| \leq \epsilon|U|$ for each i (giving the upper bound on the false positive rate)[6] and for any set $A \subseteq U$ of size n, $A \subseteq S_i$ for at least one S_i (allowing us to store A).

Let $N = |U|$. Then each set S_i covers $\binom{\epsilon N}{n}$ of the $\binom{N}{n}$ subsets of size n. If we could get them to overlap optimally (we can't), we'd still need a minimum of $\binom{N}{n} \Big/ \binom{\epsilon N}{n} = (N)_n/(\epsilon N)_n \approx (1/\epsilon)^n$ sets to cover everybody, where the approximation assumes $N \gg n$. Taking the log gives $\lg M \approx n \lg(1/\epsilon)$, meaning we need about $\lg(1/\epsilon)$ bits per key for the data structure. Bloom filters use $1/\ln 2$ times this.

There are known data structures that approach this bound asymptotically; see Pagh *et al.* [PPR05]. These also have other desirable properties, like supporting deletions and faster lookups if we can't look up bits in parallel. As far as I know, they are not used much in practice.

7.6.4 Applications

Bloom filters are popular in networking and database systems because they can be used as a cheap test to see if some key is actually present in a data structure that it's otherwise expensive to search in. Bloom filters are particular nice in hardware implementations, since the k hash functions can be computed in parallel.

An example is the **Bloomjoin** in distributed databases [ML86]. Here we want to do a join on two tables stored on different machines (a join is an operation where we find all pairs of rows, one in each table, that match on some common key). A straightforward but expensive way to do this is to send the list of keys from the smaller table across the network, then match

[6] Technically, this gives a weaker bound on false positives. For standard Bloom filters, assuming random hash functions, each key individually has at most an ϵ probability of appearing as a false positive. The hypothetical data structure we are considering here—which is effectively deterministic—allows the set of false positives to depend directly on the set of keys actually inserted in the data structure, so in principle the adversary could arrange for a specific key to appear as a false positive with probability 1 by choosing appropriate keys to insert. So this argument may underestimate the space needed to get make the false positives less predictable. On the other hand, we aren't charging the Bloom filter for the space needed to store the hash functions, which could be quite a bit if they are genuine random functions.

them against the corresponding keys from the larger table. If there are n_s rows in the smaller table, n_b rows in the larger table, and j matching rows in the larger table, this requires sending n_s keys plus j rows. If instead we send a Bloom filter representing the set of keys in the smaller table, we only need to send $\lg(1/\epsilon)/\ln 2$ bits for the Bloom filter plus an extra ϵn_b rows on average for the false positives. This can be cheaper than sending full keys across if the number of false positives is reasonably small.

7.6.5 Counting Bloom filters

It's not hard to modify a Bloom filter to support deletion. The basic trick is to replace each bit with a counter, so that whenever a value x is inserted, we increment $A[h_i(x)]$ for all i and when it is deleted, we decrement the same locations. The search procedure now returns $\min_i A[h_i(x)]$ (which means that it principle it can even report back multiplicities, though with some probability of reporting a value that is too high). To avoid too much space overhead, each array location is capped at some small maximum value c; once it reaches this value, further increments have no effect. The resulting structure is called a **counting Bloom filter**, due to Fan *et al.* [FCAB00].

We'd only expect this to work if our chances of hitting the cap is small. Fan *et al.* observe that the probability that the m table entries include one that is at least c after n insertions is bounded by

$$\begin{aligned} m\binom{nk}{c}\frac{1}{m^c} &\le m\left(\frac{enk}{c}\right)^c\frac{1}{m^c} \\ &= m\left(\frac{enk}{cm}\right)^c \\ &= m(ek\alpha/c)^c. \end{aligned}$$

(This uses the bound $\binom{n}{k} \le \left(\frac{en}{k}\right)^k$, which follows from Stirling's formula.)

For $k = \frac{1}{\alpha}\ln 2$, this is $m(e\ln 2/c)^c$. For the specific value of $c = 16$ (corresponding to 4 bits per entry), they compute a bound of $1.37\times 10^{-15}m$, which they argue is minuscule for all reasonable values of m (it's a systems paper).

The possibility that a long chain of alternating insertions and deletions might produce a false negative due to overflow is considered in the paper, but the authors state that "the probability of such a chain of events is so low that it is much more likely that the proxy server would be rebooted in the meantime and the entire structure reconstructed." An alternative way of dealing with this problem is to never decrement a maxed-out register;

this never produces a false negative, but may cause the filter to slowly fill up with maxed-out registers, producing a higher false-positive rate.

A fancier variant of this idea is the **spectral Bloom filter** [CM03], which uses larger counters to track multiplicities of items. The essential idea here is that we can guess that the number of times a particular value x was inserted is equal to $\min_{i=1}^{m} A[h_i(x)]$), with some extra tinkering to detect errors based on deviations from the typical joint distribution of the $A[h_i(x)]$ values. A fancier version of this idea gives the count-min sketches of the next section.

7.6.6 Count-min sketches

Count-min sketches are designed for use in **data stream computation**. In this model, we are given a huge flood of data—far too big to store—in a single pass, and want to incrementally build a small data structure, called a **sketch**, that will allow us to answer statistical questions about the data after we've processed it all. The motivation is the existence of data sets that are too large to store at all (network traffic statistics), or too large to store in fast memory (very large database tables). By building a sketch we can make one pass through the data set but answer queries after the fact, with some loss of accuracy.

An example of a problem in this model is that we are presented with a sequence of pairs (i_t, c_t) where $1 \leq i_t \leq n$ is an *index* and c_t is a *count*, and we want to construct a sketch that will allows us to approximately answer statistical queries about the vector a given by $a_i = \sum_{t,i[t]=i} c_t$. The size of the sketch should be polylogarithmic in the size of a and the length of the stream, and polynomial in the error bounds. Updating the sketch given a new data point should be cheap.

A solution to this problem is given by the **count-min sketch** of Cormode and Muthukrishnan [CM05] (see also [MU05, §13.4]). This gives approximations of a_i, $\sum_{i=\ell}^{r} a_i$, and $a \cdot b$ (for any fixed b), and can be used for more complex tasks like finding **heavy hitters**—indices with high weight. The easiest case is approximating a_i when all the c_t are non-negative, so we'll start with that.

7.6.6.1 Initialization and updates

To construct a count-min sketch, build a two-dimensional array c with depth $d = \lceil \ln(1/\delta) \rceil$ and width $w = \lceil e/\epsilon \rceil$, where ϵ is the error bound and δ is the probability of exceeding the error bound. Choose d independent hash

functions from some 2-universal hash family; we'll use one of these hash function for each row of the array. Initialize c to all zeros.

The update rule: Given an update (i_t, c_t), increment $c[j, h_j(i_t)]$ by c_t for $j = 1 \ldots d$. (This is the *count* part of count-min.)

7.6.6.2 Queries

Let's start with **point queries**. Here we want to estimate a_i for some fixed i. There are two cases, depending on whether the increments are all non-negative, or arbitrary. In both cases we will get an estimate whose error is linear in both the error parameter ϵ and the ℓ_1-norm $\|a\|_1 = \sum_i |a_i|$ of a. It follows that the relative error will be low for heavy points, but we may get a large relative error for light points (and especially large for points that don't appear in the data set at all).

For the non-negative case, to estimate a_i, compute $\hat{a}_i = \min_j c[j, h_j(i)]$. (This is the *min* part of coin-min.) Then:

Lemma 7.6.1. *When all c_t are non-negative, for $\hat{a}_i$ as defined above:*

$$\hat{a}_i \geq a_i, \tag{7.6.2}$$

and

$$\Pr\left[\hat{a}_i \leq a_i + \epsilon\|a\|_1\right] \geq 1 - \delta. \tag{7.6.3}$$

Proof. The lower bound is easy. Since for each pair (i, c_t) we increment each $c[j, h_j(i)]$ by c_t, we have an invariant that $a_i \leq c[j, h_j(i)]$ for all j throughout the computation, which gives $a_i \leq \hat{a}_i = \min_j c[j, h_j(i)]$.

For the upper bound, let I_{ijk} be the indicator for the event that $(i \neq k) \wedge (h_j(i) = h_j(k))$, i.e., that we get a collision between i and k using h_j. The 2-universality property of the h_j gives $\mathrm{E}\left[I_{ijk}\right] \leq 1/w \leq \epsilon/e$.

Now let $X_{ij} = \sum_{k=1}^{n} I_{ijk} a_k$. Then $c[j, h_j(i)] = a_i + X_{ij}$. (The fact that $X_{ij} \geq 0$ gives an alternate proof of the lower bound.) Now use linearity of expectation to get

$$\begin{aligned}
\mathrm{E}\left[X_{ij}\right] &= \mathrm{E}\left[\sum_{k=1}^{n} I_{ijk} a_k\right] \\
&= \sum_{k=1}^{n} a_k \,\mathrm{E}\left[I_{ijk}\right] \\
&\leq \sum_{k=1}^{n} a_k (\epsilon/e) \\
&= (\epsilon/e)\|a\|_1.
\end{aligned}$$

So $\Pr[c[j, h_j(i)] > a_i + \epsilon\|a\|_1] = \Pr[X_{ij} > e\,\mathrm{E}[X_{ij}]] < 1/e$, by Markov's inequality. With d choices for j, and each h_j chosen independently, the probability that every count is too big is at most $(1/e)^{-d} = e^{-d} \leq \exp(-\ln(1/\delta)) = \delta$. □

Now let's consider the general case, where the increments c_t might be negative. We still initialize and update the data structure as described in §7.6.6.1, but now when computing $\hat{a}_i$, we use the median count instead of the minimum count: $\hat{a}_i = \mathrm{median}\{c[j, h_j(i)] \mid j = 1 \ldots n\}$. Now we get:

Lemma 7.6.2. *For $\hat{a}_i$ as defined above,*

$$\Pr[a_i - 3\epsilon\|a\|_1 \leq \hat{a}_i \leq a_i + 3\epsilon\|a\|_1] > 1 - \delta^{1/4}. \tag{7.6.4}$$

Proof. We again define the error term X_{ij} as above, and observe that

$$\begin{aligned}
\mathrm{E}[|X_{ij}|] &= \mathrm{E}\left[\left|\sum_k I_{ijk} a_k\right|\right] \\
&\leq \sum_{k=1}^{n} |a_k \,\mathrm{E}[I_{ijk}]| \\
&\leq \sum_{k=1}^{n} |a_k|(\epsilon/e) \\
&= (\epsilon/e)\|a\|_1.
\end{aligned}$$

Using Markov's inequality, we get $\Pr[|X_{ij}|] > 3\epsilon\|a\|_1] = \Pr[|X_{ij}| > 3e\,\mathrm{E}[X_{ij}]] < 1/3e < 1/8$. In order for the median to be off by more than $3\epsilon\|a\|_1$, we need $d/2$ of these low-probability events to occur. The expected number that occur is $\mu = d/8$, so applying the standard Chernoff bound (5.2.1) with $\delta = 3$ we are looking at

$$\begin{aligned}
\Pr[S \geq d/2] &= \Pr[S \geq (1+3)\mu] \\
&\leq (e^3/4^4)^{d/8} \\
&\leq (e^{3/8}/2)^{\ln(1/\delta)} \\
&= \delta^{\ln 2 - 3/8} \\
&< \delta^{1/4}
\end{aligned}$$

(the actual exponent is about 0.31, but 1/4 is easier to deal with). This immediately gives (7.6.4). □

One way to think about this is that getting an estimate within $\epsilon\|a\|_1$ of the right value with probability at least $1-\delta$ requires 3 times the width and 4 times the depth—or 12 times the space and 4 times the time—when we aren't assuming increments are non-negative.

Next, we consider inner products. Here we want to estimate $a \cdot b$, where a and b are both stored as count-min sketches using the same hash functions. The paper concentrates on the case where a and b are both non-negative, which has applications in estimating the size of a join in a database. The method is to estimate $a \cdot b$ as $\min_j \sum_{k=1}^{w} c_a[j,k] \cdot c_b[j,k]$.

For a single j, the sum consists of both good values and bad collisions; we have $\sum_{k=1}^{w} c_a[j,k] \cdot c_b[j,k] = \sum_{k=1}^{n} a_i b_i + \sum_{p \neq q, h_j(p)=h_j(q)} a_p b_q$. The second term has expectation

$$\begin{aligned}\sum_{p \neq q} \Pr\left[h_j(p) = h_j(q)\right] a_p b_q &\leq \sum_{p \neq q} (\epsilon/e) a_p b_q \\ &\leq \sum_{p,q} (\epsilon/e) a_p b_q \\ &\leq (\epsilon/e)\|a\|_1 \|b\|_1.\end{aligned}$$

As in the point-query case, we get probability at most $1/e$ that a single j gives a value that is too high by more than $\epsilon\|a\|_1\|b\|_1$, so the probability that the minimum value is too high is at most $e^{-d} \leq \delta$.

7.6.6.3 Finding heavy hitters

Here we want to find the heaviest elements in the set: those indices i for which a_i exceeds $\phi\|a\|_1$ for some constant threshold $0 < \phi \leq 1$.

The easy case is when increments are non-negative (for the general case, see the paper), and uses a method from a previous paper by Charikar *et al.* [CCFC04]. Because $\|a\|_1 = \sum_i a_i$, we know that there will be at most $1/\phi$ heavy hitters. But the tricky part is figuring out which elements they are.

Instead of trying to find the elements after the fact, we extend the data structure and update procedure to track all the heavy elements found so far (stored in a heap), as well as $\|a\|_1 = \sum c_t$. When a new increment (i, c) comes in, we first update the count-min structure and then do a point query on a_i; if $\hat{a}_i \geq \phi\|a\|_1$, we insert i into the heap, and if not, we delete i along with any other value whose stored point-query estimate has dropped below threshold.

The trick here is that the threshold $\phi\|a\|_1$ only increases over time (remember that we are assuming non-negative increments). So if some element

i is below threshold at time t, it can only go above threshold if it shows up again, and we have a probability of at least $1 - \delta$ of including it then. This means that every heavy hitter appears in the heap with probability at least $1 - \delta$.

The total space cost for this data structure is the cost of the count-min structure plus the cost of the heap; this last part will be $O((1 + \epsilon)/\phi)$ with reasonably high probability, since this is the maximum number of elements that have weight at least $\phi\|a\|_1/(1 + \epsilon)$, the minimum needed to get an apparent weight of $\phi\|a\|_1$ even after taking into account the error in the count-min structure.

7.7 Locality-sensitive hashing (not covered Fall 2014)

Locality-sensitive hashing was invented by Indyk and Motwani [IM98] to solve the problem of designing a data structure that finds approximate nearest neighbors to query points in high dimension. We'll mostly be following this paper in this section, concentrating on the hashing parts.

7.7.1 Approximate nearest neighbor search

In the **nearest neighbor search** problem (**NNS** for short), we are given a set of n points P in a metric space with distance function d, and we want to construct a data structure that allows us to quickly find the closet point in P to any given query point. Indyk and Motwani were particularly interested in what happens in $\mathbb{R}^d$ for high dimension d under various natural metrics. Because the volume of a ball in a high-dimensional space grows exponentially with the dimension, this problem suffers from the **curse of dimensionality**[Bel57]: simple techniques based on, for example, assigning points in P to nearby locations in a grid may require searching exponentially many grid locations. Indyk and Motwani deal with this through a combination of randomization and solving the weaker problem of **ϵ-nearest neighbor search** (**ϵ-NNS**), where it's OK to return a different point p' as long as $d(q, p') \leq (1 + \epsilon) \min_{p \in P} d(q, p)$.

This problem can be solved by reduction to a simpler problem called **ϵ-point location in equal balls** or **ϵ-PLEB**. In this problem, we are given n radius-r balls centered on points in a set C, and we want a data structure that returns a point $c' \in C$ with $d(q, c') \leq (1 + \epsilon)r$ if there is at least one point c with $d(q, c) \leq r$. If there is no such point, the data structure may or may not return a point (it's allowed to say no).

The easy reduction is to use binary search. Let $R = \frac{\max_{x,y \in P} d(x,y)}{\min_{x,y \in P, x \neq y} d(x,y)}$. Given a point q, look for the minimum $\ell \in \{(1+\epsilon)^0, (1+\epsilon)^1, \ldots, R\}$ for which an ϵ-PLEB data structure with radius ℓ and centers P returns a point p with $d(q,p) \leq (1+\epsilon)\ell$; then return this point as the approximate nearest neighbor.

This requires $O(\log_{1+\epsilon} R)$ instances of the ϵ-PLEB data structure and $O(\log \log_{1+\epsilon} R)$ queries. The blowup as a function of R can be avoided using a more sophisticated data structure called a **ring-cover tree**, defined in the paper. We won't talk about ring-cover trees because they are (a) complicated and (b) not randomized. Instead, we'll move directly to the question of how we solve ϵ-PLEB.

7.7.1.1 Locality-sensitive hash functions

Definition 7.7.1 ([IM98])**.** *A family of hash functions H is (r_1, r_2, p_1, p_2)-**sensitive** for d if, for any points p and q, if h is chosen uniformly from H,*

1. *If $d(p,q) \leq r_1$, then $\Pr[h(p) = h(q)] \geq p_1$, and*
2. *If $d(p,q) > r_2$, then $\Pr[h(p) = h(q)] \leq p_2$.*

These are useful if $p_1 > p_2$ and $r_1 < r_2$; that is, we are more likely to hash inputs together if they are closer. Ideally, we can choose r_1 and r_2 to build ϵ-PLEB data structures for a range of radii sufficient to do binary search as described above (or build a ring-cover tree if we are doing it right). For the moment, we will aim for an (r_1, r_2)-PLEB data structure, which returns a point within r_1 with high probability if one exists, and never returns a point farther away than r_2.

There is some similarity between locality-sensitive hashing and a more general dimension-reduction technique known as the **Johnson-Lindenstrauss theorem** [JL84]; this says that projecting n points in a high-dimensional space to $O(\epsilon^{-2} \log n)$ dimensions using an appropriate random matrix preserves ℓ_2 distances between the points to within relative error ϵ (in fact, even a random matrix with ± 1 entries is enough [Ach03]). Unfortunately, dimension reduction by itself is not enough to solve approximate nearest neighbors in sublinear time, because we may still need to search a number of boxes exponential in $O(\epsilon^{-2} \log n)$, which will be polynomial in n.

7.7.1.2 Constructing an (r_1, r_2)-PLEB

The first trick is to amplify the difference between p_1 and p_2 so that we can find a point within r_1 of our query point q if one exists. This is done in three stages: First, we concatenate multiple hash functions to drive the probability that distant points hash together down until we get few collisions: the idea here is that we are taking the AND of the events that we get collisions in the original hash function. Second, we hash our query point and target points multiple times to bring the probability that nearby points hash together up: this is an OR. Finally, we iterate the procedure to drive down any remaining probability of failure below a target probability δ: another AND.

For the first stage, let $k = \log_{1/p_2} n$ and define a composite hash function $g(p) = (h_1(p) \dots h_k(p))$. If $d(p, q) > r_2$, $\Pr[g(p) = g(q)] \leq p_2^k = p_2^{\log_{1/p_2} n} = 1/n$. Adding this up over all n points in our data structure gives us an expected 1 false match for q.

However, we may also not be able to find the correct match for q, since p_1 may not be all that much larger than p_2. For this, we do a second round of amplification, where now we are taking the OR of events we want instead of the AND of events we don't want.

Let $\ell = n^\rho$, where $\rho = \frac{\log(1/p_1)}{\log(1/p_2)} = \frac{\log p_1}{\log p_2}$, and choose hash functions $g_1 \dots g_\ell$ independently as above. To store a point p, put it in a bucket for $g_j(p)$ for each j; these buckets are themselves stored in a hash table (by hashing the value of $g_j(p)$ down further) so that they fit in $O(n)$ space. Suppose now that $d(p, q) \leq r_1$ for some p. Then

$$\begin{aligned}
\Pr[g_j(p) = g_j(q)] &\geq p_1^k \\
&= p_1^{\log_{1/p_2} n} \\
&= n^{-\frac{\log 1/p_1}{\log 1/p_2}} \\
&= n^{-\rho} \\
&= 1/\ell.
\end{aligned}$$

So by searching through ℓ independent buckets we find p with probability at least $1 - (1 - 1/\ell)^\ell = 1 - 1/e + o(1)$. We'd like to guarantee that we only have to look at $O(n^\rho)$ points (most of which we may reject) during this process; but we can do this by stopping if we see more than 2ℓ points. Since we only expect to see ℓ bad points in all ℓ buckets, this event only happens with probability $1/2$. So even adding it to the probability of failure from the hash functions not working right we still have only a constant probability of failure $1/e + 1/2 + o(1)$.

Iterating the entire process $O(\log(1/\delta))$ times then gives the desired bound δ on the probability that this process fails to find a good point if one exists.

Multiplying out all the costs gives a cost of a query of $O(k\ell \log(1/\delta)) = O\left(n^\rho \log_{1/p_2} n \log(1/\delta)\right)$ hash function evaluations and $O(n^\rho \log(1/\delta))$ distance computations. The cost to insert a point is just $O(k\ell \log(1/\delta)) = O\left(n^\rho \log_{1/p_2} n \log(1/\delta)\right)$ hash function evaluations, the same number as for a query.

7.7.1.3 Hash functions for Hamming distance

Suppose that our points are d-bit vectors and that we use Hamming distance for our metric. In this case, using the family of one-bit projections $\{h_i \mid h_i(x) = x_i\}$ gives a locality-sensitive hash family [ABMRT96].

Specifically, we can show this family is $(r, r(1+\epsilon), 1-\frac{r}{d}, 1-\frac{r(1+\epsilon)}{d})$-sensitive. The argument is trivial: if two points p and q are at distance r or less, they differ in at most r places, so the probability that they hash together is just the probability that we don't pick one of these places, which is at least $1-\frac{r}{d}$. Essentially the same argument works when p and q are far away.

These are not particularly clever hash functions, so the heavy lifting will be done by the (r_1, r_2)-PLEB construction. Our goal is to build an ϵ-PLEB for any fixed r, which will correspond to an $(r, r(1+\epsilon))$-PLEB. The main thing we need to do, following [IM98] as always, is compute a reasonable bound on $\rho = \frac{\log p_1}{\log p_2} = \frac{\ln(1-r/d)}{\ln(1-(1+\epsilon)r/d)}$. This is essentially just a matter of hitting it with enough inequalities, although there are a couple of tricks in the middle.

Compute

$$\begin{aligned}
\rho &= \frac{\ln(1-r/d)}{\ln(1-(1+\epsilon)r/d)} \\
&= \frac{(d/r)\ln(1-r/d)}{(d/r)\ln(1-(1+\epsilon)r/d)} \\
&= \frac{\ln((1-r/d)^{d/r})}{\ln((1-(1+\epsilon)r/d)^{d/r})} \\
&\leq \frac{\ln(e^{-1}(1-r/d))}{\ln e^{-(1+\epsilon)}} \\
&= \frac{-1+\ln(1-r/d)}{-(1+\epsilon)} \\
&= \frac{1}{1+\epsilon} - \frac{\ln(1-r/d)}{1+\epsilon}. \qquad (7.7.1)
\end{aligned}$$

Note that we used the fact that $1+x \leq e^x$ for all x in the denominator and $(1-x)^{1/x} \geq e^{-1}(1-x)$ for $x \in [0,1]$ in the numerator. The first fact is our usual favorite inequality.

The second can be proved in a number of ways. The most visually intuitive is that $(1-x)^{1/x}$ and $e^{-1}(1-x)$ are equal at $x=1$ and equal in the limit as x goes to 0, while $(1-x)^{1/x}$ is concave in between 0 and 1 and $e^{-1}(1-x)$ is linear. Unfortunately it is rather painful to show that $(1-x)^{1/x}$ is in fact concave. An alternative is to rewrite the inequality $(1-x)^{1-x} \geq e^{-1}(1-x)$ as $(1-x)^{1/x-1} \geq e^{-1}$, apply a change of variables $y = 1/x$ to get $(1-1/y)^{y-1} \geq e^{-1}$ for $y \in [1,\infty)$, and then argue that (a) equality holds in the limit as y goes to infinity, and (b) the left-hand-side is a nonincreasing function, since

$$\begin{aligned}
\frac{d}{dy}\ln\left((1-1/y)^{y-1}\right) &= \frac{d}{dy}\left[(y-1)\left(\ln(y-1)-\ln y\right)\right] \\
&= \ln(1-1/y) + (y-1)\left(\frac{1}{y-1} - \frac{1}{y}\right) \\
&= \ln(1-1/y) + 1 - (1-1/y) \\
&= \ln(1-1/y) + 1/y \\
&\leq -1/y + 1/y \\
&= 0.
\end{aligned}$$

We now return to (7.7.1). We'd really like the second term to be small enough that we can just write n^ρ as $n^{1/(1+\epsilon)}$. (Note that even though it looks

negative, it isn't, because $\ln(1 - r/d)$ is negative.) So we pull a rabbit out of a hat by assuming that $r/d < 1/\ln n$.[7] This assumption can be justified by modifying the algorithm so that d is padded out with up to $d \ln n$ unused junk bits if necessary. Using this assumption, we get

$$\begin{aligned} n^\rho &< n^{1/(1+\epsilon)} n^{-\ln(1-1/\ln n)/(1+\epsilon)} \\ &= n^{1/(1+\epsilon)} (1 - 1/\ln n)^{-\ln n} \\ &\leq e n^{1/(1+\epsilon)}. \end{aligned}$$

Plugging into the formula for (r_1, r_2)-PLEB gives $O(n^{1/(1+\epsilon)} \log n \log(1/\delta))$ hash function evaluations per query, each of which costs $O(1)$ time, plus $O(n^{1/(1+\epsilon)} \log(1/\delta))$ distance computations, which will take $O(d)$ time each. If we add in the cost of the binary search, we have to multiply this by $O(\log \log_{1+\epsilon} R \log \log \log_{1+\epsilon} R)$, where the log-log-log comes from having to adjust δ so that the error doesn't accumulate too much over all $O(\log \log R)$ steps. The end result is that we can do approximate nearest-neighbor queries in

$$O\left(n^{1/(1+\epsilon)} \log(1/\delta)(\log n + d) \log \log_{1+\epsilon} R \log \log \log_{1+\epsilon} R\right)$$

time. For ϵ reasonably large, this is much better than naively testing against all points in our database, which takes $O(nd)$ time (but produces an exact result).

7.7.1.4 Hash functions for ℓ_1 distance

Essentially the same approach works for (bounded) ℓ_1 distance, using **discretization**, where we replace a continuous variable over some range with a discrete variable. Suppose we are working in $[0, 1]^d$ with the ℓ_1 metric. Represent each coordinate x_i as a sequence of d/ϵ values x_{ij} in **unary**, for $j = 1 \ldots \epsilon d$, with $x_{ij} = 1$ if $\epsilon j/d < x_i$. Then the Hamming distance between the bit-vectors representing x and y is proportional to the ℓ_1 distance between the original vectors, plus an error term that is bounded by ϵ. We can then use the hash functions for Hamming distance to get a locality-sensitive hash family.

A nice bit about this construction is that we don't actually have to build the bit-vectors; instead, we can specify a coordinate x_i and a threshold c and get the same effect by recording whether $x_i > c$ or not.

[7] Indyk and Motwani pull this rabbit out of a hat a few steps earlier, but it's pretty much the same rabbit either way.

Note that this does increase the cost slightly: we are converting d-dimensional vectors into (d/ϵ)-long bit vectors, so the $\log(n+d)$ term becomes $\log(n+d/\epsilon)$. When n is small, this effectively multiples the cost of a query by an extra $\log(1/\epsilon)$. More significant is that we have to cut ϵ in half to obtain the same error bounds, because we now pay ϵ error for the data structure itself and an additional ϵ error for the discretization. So our revised cost for the ℓ_1 case is

$$O\left(n^{1/(1+\epsilon/2)} \log(1/\delta)(\log n + d/\epsilon) \log\log_{1+\epsilon/2} R \log\log\log_{1+\epsilon/2} R\right).$$

Chapter 8

Martingales and stopping times

In §5.3.2, we used martingales to show that the outcome of some process was tightly concentrated. Here we will show how martingales interact with **stopping times**, which are random variables that control when we stop carrying out some task. This will require a few new definitions.

The general form of a martingale $\{X_t, \mathcal{F}_t\}$ consists of:

- A sequence of random variables $X_0, X_1, X_2, \ldots$; and
- A **filtration** $\mathcal{F}_0 \subseteq \mathcal{F}_1 \subseteq \mathcal{F}_2 \ldots$, where each σ-algebra $\mathcal{F}_t$ represents our knowledge at time t;

subject to the requirements that:

1. The sequence of random variables is **adapted** to the filtration, which just means that each X_t is measurable $\mathcal{F}_t$ or equivalently that $\mathcal{F}_t$ (and thus all subsequent $\mathcal{F}_{t'}$ for $t' \geq t$) includes all knowledge of X_t; and
2. The **martingale property**

$$\mathrm{E}\left[X_{t+1} \mid \mathcal{F}_t\right] = X_t \tag{8.0.1}$$

   holds for all t.

Together with this more general definition of a martingale, we will also use the following definition of a **stopping time**. Given a filtration $\{\mathcal{F}_t\}$, a random variable τ is a stopping time for $\{\mathcal{F}_t\}$ if $\tau \in \mathbb{N} \cup \{\infty\}$ and the event

$[\tau \leq t]$ is measurable $\mathcal{F}_t$ for all t.[1] In simple terms, τ is a stopping time if you know at time t whether to stop there or not.

What we like about martingales is that iterating the martingale property shows that $\mathrm{E}[X_t] = \mathrm{E}[X_0]$ for all fixed t. We will show that, under reasonable conditions, the same holds for X_τ when τ is a stopping time. (The random variable X_τ is defined in the obvious way, as a random variable that takes on the value of X_t when $\tau = t$.)

8.1 Submartingales and supermartingales

In some cases we have a process where instead of getting equality in (8.0.1), we get an inequality instead. A **submartingale** replaces (8.0.1) with

$$X_t \leq \mathrm{E}[X_{t+1} \mid \mathcal{F}_t] \tag{8.1.1}$$

while a **supermartingale** satisfies

$$X_t \geq \mathrm{E}[X_{t+1} \mid \mathcal{F}_t]. \tag{8.1.2}$$

In each case, what is "sub" or "super" is the value at the current time compared to the expected value at the next time. Intuitively, a submartingale corresponds to a process where you win on average, while a supermartingale is a process where you lose on average. Casino games (in profitable casinos) are submartingales for the house and supermartingales for the player.

Sub- and supermartingales can be reduced to martingales by subtracting off the expected change at each step. For example, if $\{X_t\}$ is a submartingale with respect to $\{\mathcal{F}_t\}$, then the process $\{Y_t\}$ defined recursively by

$$\begin{aligned} Y_0 &= X_0 \\ Y_{t+1} &= Y_t + X_{t+1} - \mathrm{E}[X_{t+1} \mid \mathcal{F}_t] \end{aligned}$$

is a martingale, since

$$\begin{aligned} \mathrm{E}[Y_{t+1} \mid \mathcal{F}_t] &= \mathrm{E}[Y_t + X_{t+1} - \mathrm{E}[X_{t+1} \mid \mathcal{F}_t] \mid \mathcal{F}_t] \\ &= Y_t + \mathrm{E}[X_{t+1} \mid \mathcal{F}_t] - \mathrm{E}[X_{t+1} \mid \mathcal{F}_t] \\ &= 0. \end{aligned}$$

[1] Different authors impose different conditions on the range of τ; for example, Mitzenmacher and Upfal [MU05] exclude the case $\tau = \infty$. We allow $\tau = \infty$ to represent the outcome where we never stop. This can be handy for modeling processes where this outcome is possible, although in practice we will typically insist that it occurs only with probability zero.

One way to think of this is that $Y_t = X_t + \delta_t$, where δ_t is a predictable, non-decreasing **drift process** that starts at 0. For supermartingales, the same result holds, but now δ_t is non-increasing. This ability to decompose an adapted stochastic process into the sum of a martingale and a predictable drift process is known as the **Doob decomposition theorem**.

8.2 The optional stopping theorem

If $(X_t, \mathcal{F}_t)$ is a martingale, then applying induction to the martingale property shows that $\mathrm{E}[X_t] = \mathrm{E}[X_0]$ for any fixed time t. The **optional stopping theorem** shows that this also happens for X_τ when τ is a stopping time, under various choices of additional conditions:

Theorem 8.2.1. *Let $(X_t, \mathcal{F}_t)$ be a martingale and τ a stopping time for $\{\mathcal{F}_t\}$. Then $\mathrm{E}[X_\tau] = E[X_0]$ if at least one of the following conditions holds:*

1. ***Bounded time.*** *There is a fixed n such that $\tau \leq n$ always.*
2. ***Bounded range and finite time.*** *There is a fixed M such that $|X_t| \leq M$ always, and $\Pr[\tau < \infty] = 1$.*
3. ***Bounded increments and finite expected time.*** *There is a fixed c such that $|X_{t+1} - X_t| \leq c$ always, and $\mathrm{E}[\tau] < \infty$.*
4. ***General case.*** *All three of the following conditions hold:*
   (a) $\Pr[\tau < \infty] = 1$,
   (b) $\mathrm{E}[|X_\tau|] < \infty$, *and*
   (c) $\lim_{t\to\infty} \mathrm{E}\left[X_t \cdot 1_{[\tau > t]}\right] = 0$.

It would be nice if we could show $\mathrm{E}[X_\tau] = E[X_0]$ without the side conditions, but in general this isn't true. For example, the double-after-losing martingale strategy in the St. Petersburg paradox (see §3.4.1.1) eventually yields $+1$ with probability 1, so if τ is the time we stop playing, we have $\Pr[\tau < \infty] = 1$, $\mathrm{E}[|X_\tau|] < \infty$, but $\mathrm{E}[X_\tau] = 1 \neq \mathrm{E}[X_0] = 0$. But in order for this to happen, we have to violate all of bounded time (τ is not bounded), bounded range ($|X_t|$ roughly doubles every step until we stop), bounded increments ($|X_{t+1} - X_t|$ doubles every step as well), or one of the three conditions of the general case (the last one: $\lim_{t\to\infty} \mathrm{E}[X_t] \cdot 1_{[\tau > t]} = -1 \neq 0$).

The next section gives a proof of the optional stopping theorem. The intuition is that for any fixed n, we can **truncate** X_τ to $X_{\min(\tau,n)}$ and show

that $\mathrm{E}\left[X_{\min(\tau,n)}\right] = \mathrm{E}\left[X_0\right]$. This immediately gives the bounded time case. For the other cases, the argument is that $\lim_{n\to\infty} \mathrm{E}\left[X_{\min(\tau,n)}\right]$ converges to $\mathrm{E}\left[X_\tau\right]$ provided the missing part $\mathrm{E}\left[X_\tau - X_{\min(\tau,n)}\right]$ converges to zero. How we do this depends on which assumptions we are making.

A proof of the optional stopping theorem is given in the next section. You probably don't need to understand this proof to apply the theorem, and may wish to skip directly to applications in §8.4.

8.3 Proof of the optional stopping theorem (optional)

We start with the finite-time case, which we can enforce by truncating the actual process.

Lemma 8.3.1. *Let $(X_t, \mathcal{F}_t)$ be a martingale and τ a stopping time for $\{\mathcal{F}_t\}$. Then for any $n \in \mathbb{N}$, $\mathrm{E}\left[X_{\min(\tau,n)}\right] = E[X_0]$.*

Proof. Define $Y_t = X_0 + \sum_{i=1}^{t}(X_t - X_{t-1})1_{[\tau>t-1]}$. Then $(Y_t, \mathcal{F}_t)$ is a martingale, because we can calculate $\mathrm{E}\left[Y_{t+1} \mid \mathcal{F}_t\right] = \mathrm{E}\left[Y_t + (X_{t+1} - X_t)1_{[\tau>t]} \,\middle|\, \mathcal{F}_t\right] = Y_t + 1_{[\tau>t]} \cdot \mathrm{E}\left[X_{t+1} - X_t \mid \mathcal{F}_t\right] = Y_t$; effectively, we are treating $1_{[\tau\le t-1]}$ as a sequence of bets, and we know that adjusting our bets doesn't change the martingale property. But then $\mathrm{E}\left[X_{\min(\tau,n)}\right] = \mathrm{E}\left[Y_n\right] = \mathrm{E}\left[Y_0\right] = \mathrm{E}\left[X_0\right]$. □

This gives us the bounded-time variant for free: If $\tau \le n$ always, then $X_\tau = X_{\min(\tau,n)}$, and $\mathrm{E}\left[X_\tau\right] = \mathrm{E}\left[X_{\min(\tau,n)}\right] = \mathrm{E}\left[X_0\right]$.

For the unbounded-time variants, we will apply some version of the following strategy:

1. Observe that since $\mathrm{E}\left[X_{\min(\tau,n)}\right] = \mathrm{E}\left[X_0\right]$ is a constant for any fixed n, $\lim_{n\to\infty} \mathrm{E}\left[X_{\min(\tau,n)}\right]$ converges to $\mathrm{E}\left[X_0\right]$.

2. Argue that whatever assumptions we are using imply that $\lim_{n\to\infty} \mathrm{E}\left[X_{\min(\tau,n)}\right]$ also converges to $\mathrm{E}\left[X_\tau\right]$.

3. Conclude that these two limits are equal, so $\mathrm{E}\left[X_\tau\right] = \mathrm{E}\left[X_0\right]$.

For the middle step, start with

$$X_\tau = X_{\min(\tau,n)} + 1_{[\tau>n]}(X_\tau - X_n).$$

This holds because either $\tau \leq n$, and we just get X_τ, or $\tau > n$, and we get $X_n + (X_\tau - X_n) = X_\tau$.

Taking the expectation of both sides gives

$$\begin{aligned} \mathrm{E}\left[X_\tau\right] &= \mathrm{E}\left[X_{\min(\tau,n)}\right] + \mathrm{E}\left[1_{[\tau>n]}(X_\tau - X_n)\right]. \\ &= \mathrm{E}\left[X_0\right] + \mathrm{E}\left[1_{[\tau>n]}(X_\tau - X_n)\right]. \end{aligned}$$

So if we can show that the right-hand term goes to zero in the limit, we are done.

For the bounded-range case, we have $|X_\tau - X_n| \leq 2M$, so $\left|\mathrm{E}\left[1_{[\tau>n]}\right](X_\tau - X_n)\right| \leq 2M \cdot \Pr\left[\tau > n\right]$. Since in this case we assume $\Pr\left[\tau < \infty\right] = 1$, $\lim_{n\to\infty} \Pr\left[\tau > n\right] = 0$, and the theorem holds.

For bounded increments, we have

$$\begin{aligned} \left|\mathrm{E}\left[(X_\tau - X_n)1_{[\tau>n]}\right]\right| &= \left|\mathrm{E}\left[\sum_{t\geq n}(X_{t+1} - X_t)1_{[\tau>t]}\right]\right| \\ &\leq \mathrm{E}\left[\sum_{t\geq n}|(X_{t+1} - X_t)|1_{[\tau>t]}\right] \\ &\leq \mathrm{E}\left[\sum_{t\geq n} c1_{[\tau>t]}\right] \\ &\leq c\,\mathrm{E}\left[\sum_{t\geq n} 1_{[\tau>t]}\right]. \end{aligned}$$

But $\mathrm{E}\left[\tau\right] = \sum_{t=0}^{\infty} 1_{[\tau>t]}$. Under the assumption that this sequence converges, its tail goes to zero, and again the theorem holds.

For the general case, we can expand

$$\mathrm{E}\left[X_\tau\right] = \mathrm{E}\left[X_{\min(\tau,n)}\right] + \mathrm{E}\left[1_{[\tau>n]}X_\tau\right] - \mathrm{E}\left[1_{[\tau>n]}X_n\right]$$

Because $\min(\tau, n)$ is a stopping time bounded by n, we know from the bounded-time case that $\mathrm{E}\left[X_{\min(\tau,)}\right] = \mathrm{E}\left[X_0\right]$ for each n, and this continues to be true if we take the limit as n goes to infinity. Condition (4c) gives $\lim_{n\to\infty} \mathrm{E}\left[1_{[\tau>n]}X_n\right] \to 0$, so the last term vanishes in the limit. If we can show that the middle term also vanishes in the limit, we are done.

Here we use condition (4b). Observe that $\mathrm{E}\left[1_{[\tau>n]}X_\tau\right] = \sum_{t=n+1}^{\infty} \mathrm{E}\left[1_{[\tau=t]}X_t\right]$. Compare this with $\mathrm{E}\left[X_\tau\right] = \sum_{t=0}^{\infty} \mathrm{E}\left[1_{[\tau=t]}X_t\right]$; this is an absolutely convergent series (this is why we need condition (4b)), so in the limit the sum of

the terms for $i = 0 \ldots n$ converges to $\mathrm{E}[X_\tau]$. But this means that the sum of the remaining terms for $i = n+1 \ldots \infty$ converges to zero. So the middle term goes to zero as n goes to infinity.

We thus have

$$\begin{aligned}\mathrm{E}[X_\tau] &= \lim_{n\to\infty}\left(\mathrm{E}\left[X_{\min(\tau,n)}\right] + \mathrm{E}\left[1_{[\tau>n]}X_\tau\right] - \mathrm{E}\left[1_{[\tau>n]}X_n\right]\right)\\ &= \lim_{n\to\infty}\mathrm{E}\left[X_{\min(\tau,n)}\right]\\ &= \mathrm{E}[X_0].\end{aligned}$$

This completes the proof.

8.4 Applications

8.4.1 Random walks

Let X_t be an **unbiased** ± 1 **random walk** that starts at 0, adds ± 1 to its current position with equal probability at each step, and stops if it reaches $+a$ or $-b$.[2] We'd like to calculate the probability of reaching $+a$ before $-b$. Let τ be the time at which the process stops. We can easily show that $\Pr[\tau < \infty] = 1$ and $\mathrm{E}[\tau] < \infty$ by observing that from any state of the random walk, there is a probability of at least $2^{-(a+b)}$ that it stops within $a+b$ steps (by flipping heads $a+b$ times in a row), so that if we consider a sequence of intervals of length $a+b$, the expected number of such intervals we can have before we stop is at most 2^{a+b}. We have bounded increments by the definition of the process (bounded range also works). So $\mathrm{E}[X_\tau] = E[X_0] = 0$ and the probability p of landing on $+a$ instead of $-b$ must satisfy $pa - (1-p)b = 0$, giving $p = \frac{b}{a+b}$.

Now suppose we want to find $\mathrm{E}[\tau]$. Let $Y_t = X_t^2 - t$. Then $Y_{t+1} = (X_t \pm 1)^2 - (t+1) = X_t^2 \pm 2X_t + 1 - (t+1) = (X_t^2 - t) \pm 2X_t = Y_t \pm 2X_t$. Since the plus and minus cases are equally likely, they cancel out in expectation and $\mathrm{E}[Y_{t+1} \mid \mathcal{F}_t] = Y_t$: we just showed Y_t is a martingale.[3] We also show it has bounded increments (at least up until time τ), because $|Y_{t+1} - Y_t| = 2|X_t| \leq \max(a,b)$.

[2] This is called a **random walk with two absorbing barriers**.

[3] This construction generalizes in a nice way to arbitrary martingales. Suppose $\{X_t\}$ is a martingale with respect to $\{\mathcal{F}_t\}$. Let $\delta_t = X_t - X_{t-1}$, and let $V_t = \mathrm{Var}[\delta_t \mid \mathcal{F}_{t-1}]$ be the conditional variance of the t-th increment (note that this is a random variable that may depend on previous outcomes). We can easily show that $Y_t = X_t^2 - \sum_{i=1}^t V_i$ is a

From Theorem 8.2.1, $\mathrm{E}[Y_\tau] = 0$, which gives $\mathrm{E}[\tau] = \mathrm{E}\left[X_\tau^2\right]$. But we can calculate $\mathrm{E}\left[X_\tau^2\right]$: it is $a^2 \Pr[X_\tau = a] + b^2 \Pr[X_t = -b] = a^2(b/(a+b)) + b^2(a/(a+b)) = (a^2b + b^2a)/(a+b) = ab$.

If we have a random walk that only stops at $+a$,[4] then if τ is the first time at which $X_\tau = a$, τ is a stopping time. However, in this case $\mathrm{E}[X_\tau] = a \neq \mathrm{E}[X_0] = 0$. So the optional stopping theorem doesn't apply in this case. But we have bounded increments, so Theorem 8.2.1 apply if $\mathrm{E}[\tau] < \infty$. It follows that the expected time until we reach a is unbounded, either because sometimes we never reach a, or because we always reach a but sometimes it takes a very long time. [5]

---

martingale. The proof is that

$$\begin{aligned}
\mathrm{E}[Y_t \mid \mathcal{F}_{t-1}] &= \mathrm{E}\left[X_t^2 - \sum_{i=1}^{t} V_t \;\middle|\; \mathcal{F}_{t-1}\right] \\
&= \mathrm{E}\left[(X_{t-1} + \delta_t)^2 \mid \mathcal{F}_{t-1}\right] - \sum_{i=1}^{t} V_i \\
&= \mathrm{E}\left[X_{t-1}^2 + 2X_{t-1}\delta_t + \delta_t^2 \mid \mathcal{F}_{t-1}\right] - \sum_{i=1}^{t} V_i \\
&= X_{t-1}^2 + 2X_{t-1}\,\mathrm{E}[\delta_t \mid \mathcal{F}_{t-1}] + \mathrm{E}\left[\delta_t^2 \mid \mathcal{F}_{t-1}\right] - \sum_{i=1}^{t} V_i \\
&= X_{t-1}^2 + 0 + V_t - \sum_{i=1}^{t} V_i \\
&= X_{t-1}^2 - \sum_{i=1}^{t-1} V_i \\
&= Y_{t-1}.
\end{aligned}$$

For the ± 1 random walk case, we have $V_t = 1$ always, giving $\sum_{i=1}^{t} V_i = t$ and $\mathrm{E}\left[X_\tau^2\right] = \mathrm{E}\left[X_0^2\right] + \mathrm{E}[\tau]$ when τ is a stopping time satisfying the conditions of the Optional Stopping Theorem. For the general case, the same argument gives $\mathrm{E}\left[X_\tau^2\right] = \mathrm{E}\left[X_0^2\right] + \mathrm{E}\left[\sum_{t=1}^{\tau} V_t\right]$ instead: the expected square position of X_t is incremented by the conditional variance at each step.

[4] This would be a **random walk with one absorbing barrier**.

[5] In fact, we always reach a. An easy way to see this is to imagine a sequence of intervals of length $n_1, n_2, \ldots$, where $n_{i+1} = \left(a + \sum_{j=1}^{i} n_j\right)^2$. At the end of the i-th interval, we are no lower than $-\sum_{j=0}^{i} n_j$, so we only need to go up $\sqrt{n_{i+1}}$ positions to reach a by the end of the $(i+1)$-th interval. Since this is just one standard deviation, it occurs with constant probability, so after a finite expected number of intervals, we will reach $+a$. Since there are infinitely many intervals, we reach $+a$ with probability 1.

We can also consider a **biased random walk** where $+1$ occurs with probability $p > 1/2$ and -1 with probability $q = 1 - p$. If X_t is the position of the random walk at time t, it isn't a martingale. But $Y_t = X_t - (p - q)t$ is, and it even has bounded increments. So if τ is the time at which $X_t = a$ and $\mathrm{E}[\tau]$ is finite,[6] the optional stopping theorem gives $\mathrm{E}[Y_\tau] = E[Y_0] = 0$, which gives $\mathrm{E}[a - (p - q)t] = 0$ or $\mathrm{E}[t] = \frac{a}{p-q}$, pretty much what we'd expect.

8.4.2 Wald's equation

Suppose we run a Las Vegas algorithm until it succeeds, and the i-th attempt costs X_i, where all the X_i are independent, satisfy $0 \le X_i \le c$ for some c, and have a common mean $\mathrm{E}[X_i] = \mu$. Let N be the number of times we run the algorithm; since we can tell when we are done, N is a stopping time with respect to some filtration $\{\mathcal{F}_i\}$ to which the X_i are adapted.[7] Suppose also that $\mathrm{E}[N]$ exists. What is $\mathrm{E}\left[\sum_{i=1}^N X_i\right]$?

If N were not a stopping time, this might be a very messy problem indeed. But when N is a stopping time, we can apply it to the martingale $Y_t = \sum_{i=1}^t (X_i - \mu)$. This has bounded increments ($0 \le X_i \le c$, so $-c \le X_i - \mathrm{E}[X_i] \le c$), and we've already said $\mathrm{E}[N]$ is finite (which implies $\Pr[N < \infty] = 1$), so Theorem 8.2.1 applies. We thus have

$$\begin{aligned}
0 &= \mathrm{E}[Y_N] \\
&= \mathrm{E}\left[\sum_{i=1}^N (X_i - \mu)\right] \\
&= \mathrm{E}\left[\sum_{i=1}^N X_i\right] - \mathrm{E}\left[\sum_{i=1}^N \mu\right] \\
&= \mathrm{E}\left[\sum_{i=1}^N X_i\right] - \mathrm{E}[N]\,\mu.
\end{aligned}$$

Rearranging this gives **Wald's equation**:

$$\mathrm{E}\left[\sum_{i=1}^N X_i\right] = \mathrm{E}[N]\,\mu. \tag{8.4.1}$$

This is the same formula as in §3.4.2.1, but we've eliminated the bound on N and allowed for much more dependence between N and the X_i.

[6] Exercise: Show $\mathrm{E}[\tau]$ is finite.

[7] A stochastic process $\{X_t\}$ is **adapted** to a filtration $\{\mathcal{F}_t\}$ if each X_t is measurable $\mathcal{F}_t$.

8.4.3 Waiting times for patterns

Let's suppose we flip coins until we see some pattern appear: for example, we might flip coins until we see HTHH. What is the expected number of coin-flips until this happens?

A very clever trick due to Li [Li80] solves this problem exactly using the Optional Stopping Theorem. Suppose our pattern is $x_1 x_2 \dots x_k$. We imagine an army of gamblers, one of which shows up before each coin-flip. Each gambler starts by betting \$1 that next coin-flip will be x_1. If she wins, she bets \$2 that the next coin-flip will be x_2, continuing to play double-or-nothing until either she loses (and is \$1) or wins her last bet on x_k (and is up $2^k - 1$). Because each gambler's winnings form a martingale, so does their sum, and so the expected total return of all gamblers up to the stopping time τ at which our pattern first occurs is 0.

We can now use this fact to compute $\mathrm{E}[\tau]$. When we stop at time τ, we have one gambler who has won $2^k - 1$. We may also have other gamblers who are still in play. For each i with $x_1 \dots x_i = x_{k-i+1} \dots x_k$, there will be a gambler with net winnings $\sum_{j=1}^{i} 2^{j-1} = 2^i - 1$. The remaining gamblers will all be at -1.

Let $\chi_i = 1$ if $x_1 \dots x_i = x_{k-i+1} \dots x_k$, and 0 otherwise. Then the number of losers is given by $\tau - \sum_{i=1}^{k} \chi_i$ and the total expected payoff is

$$\begin{aligned}
\mathrm{E}[X_\tau] &= \mathrm{E}\left[-(\tau - \sum_{i=1}^{k} \chi_i) + \sum_{i=1}^{k} \chi_i \left(2^i - 1\right)\right] \\
&= \mathrm{E}\left[-\tau + \sum_{i=1}^{k} \chi_i \left(2^i\right)\right] \\
&= 0.
\end{aligned}$$

It follows that $\mathrm{E}[\tau] = \sum_{i=1}^{k} \chi_i 2^i$.

As a quick test, the pattern H has $\mathrm{E}[\tau] = 2^1 = 2$. This is consistent with what we know about geometric distributions.

For a longer example, the pattern HTHH only overlaps with its prefix H so in this case we have $\mathrm{E}[\tau] = \sum \chi_i 2^i = 16 + 2 = 18$. But HHHH overlaps with all of its prefixes, giving $\mathrm{E}[\tau] = 16 + 8 + 4 + 2 = 30$. At the other extreme, THHH has no overlap at all and gives $\mathrm{E}[\tau] = 16$.

In general, for a pattern of length k, we expect a waiting time somewhere between 2^k and $2^{k+1} - 2$—almost a factor of 2 difference depending on how much overlap we get.

This analysis generalizes in the obvious way to biased coins and larger alphabets; see the paper [Li80] for details.

Chapter 9

Markov chains

A **stochastic process** is a sequence of random variables $\{X_t\}$, where we think of X_t as the value of the process at time t.

There are two stochastic processes that come up over and over again in the analysis of randomized algorithms. One is a martingale, where the next increment may depend in a complicated way on the past history but has expectation 0; the other is a Markov chain, where the next step depends only on the current location and not the previous history. We've already seen martingales before in §5.3.2 and Chapter 8. In this chapter, we'll give basic definitions for Markov chains, and then talk about the most useful algorithmic property, which is convergence to a fixed distribution on states after sufficiently many steps in many Markov chains.

If you want to learn more about Markov chains than presented here or in the textbook, they are usually covered in general probability textbooks (for example, in [Fel68] or [GS01]), mentioned in many linear algebra textbooks [Str03], covered in some detail in stochastic processes textbooks [KT75], and covered in exquisite detail in many books dedicated specifically to the subject [KS76, KSK76]. For reversible Markov chains and Markov chains arising from random walks on graphs, the legendary Aldous-fill manuscript is worth looking at [AF01].

9.1 Basic definitions and properties

A **Markov chain** or **Markov process** is a stochastic process where the distribution of X_{t+1} depends only on the value of X_t and not any previous

history. Formally, this means that

$$\Pr\left[X_{t+1} = j \mid X_t = i_t, X_{t-1} = i_{t-1}, \ldots, X_0 = i_0\right] = \Pr\left[X_{t+1} = j \mid X_t = i_t\right]. \tag{9.1.1}$$

A stochastic process with this property is called **memoryless**: at any time, you know where you are, and you can figure out where you are going, but you don't know where you were before.

The **state space** of the chain is just the set of all values that each X_t can have. A Markov chain is **finite** or **countable** if it has a finite or countable state space, respectively. We'll mostly be interested in countable Markov chains.[1]

We'll also assume that our Markov chains are **homogeneous**, which means that $\Pr\left[X_{t+1} = j \mid X_t = i\right]$ doesn't depend on t.

For a homogeneous countable Markov chain, we can describe its behavior completely by giving the state space and the one-step **transition probabilities** $p_{ij} = \Pr\left[X_{t+1} = j \mid X_t = i\right]$. Given p_{ij}, we can calculate two-step transition probabilities

$$\begin{aligned} p_{ij}^{(2)} &= \Pr\left[X_{t+2} = j \mid X_t = i\right] \\ &= \sum_k \Pr\left[X_{t+2} = j \mid X_{t+1} = k\right] \Pr\left[X_{t+1} = k \mid X_t = i\right] \\ &= \sum_k p_{ik} p_{kj}. \end{aligned}$$

This is identical to the formula for matrix multiplication, and so if the transition probabilities are given by a transition matrix P, then $p_{ij}^{(2)} = (P^2)_{ij}$ and in general the n-step transition probability $p_{ij}^{(n)} = (P^n)_{ij}$.

Conversely, given any matrix with non-negative entries where the rows sum to 1 ($\sum_j P_{ij} = 1$, or $P\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ in the second equation stands for the all-ones vector), there is a corresponding Markov chain given by $p_{ij} = P_{ij}$. Such a matrix is called a **stochastic matrix**.

The general formula for $(n+m)$-step transition probabilities is that $p_{ij}^{(n+m)} = \sum_k p_{ik}^{(n)} p_{kj}^{(m)}$. This is known as the **Chapman-Kolmogorov equation** and is equivalent to the matrix identity $P^{n+m} = P^n P^m$.

[1] If the state space is not countable, we run into the same measure-theoretic issues as with continuous random variables, and have to replace (9.1.1) with the more general condition that

$$\mathrm{E}\left[1_{[X_t \in A]} \;\middle|\; X_t, X_{t-1}, \ldots, X_0\right] = \mathrm{E}\left[1_{[X_t \in A]} \;\middle|\; X_t\right],$$

provided A is measurable with respect to some appropriate σ-algebra. We don't really want to deal with this, and for the most part we don't have to, so we won't.

A distribution over states of the Markov chain at some time t can be given by a row vector x, where $x_i = Pr[X_t = i]$. To compute the distribution at time $t+1$, we use the law of total probability: $\Pr[X_{t+1} = j] = \sum_i \Pr[X_t = i] \Pr[X_{t+1} = j \mid X_t = i] = \sum_i x_i p_{ij}$. Again we have the formula for matrix multiplication (where we treat x as a $1 \times i$ matrix); so the distribution vector at time $t+1$ is just xP, and at time $t+n$ is xP^n.

We like Markov chains for two reasons:

1. They describe what happens in a randomized algorithm; the state space is just the set of all states of the algorithm, and the Markov property holds because the algorithm can't remember anything that isn't part of its state. So if we want to analyze randomized algorithms, we will need to get good at analyzing Markov chains.

2. They can be used to do sampling over interesting distributions. Under appropriate conditions (see below), the state of a Markov chain converges to a **stationary distribution**. If we build the right Markov chain, we can control what this stationary distribution looks like, run the chain for a while, and get a sample close to the stationary distribution.

In both cases we want to have a bound on how long it takes the Markov chain to converge, either because it tells us when our algorithm terminates, or because it tells us how long to mix it up before looking at the current state.

9.1.1 Examples

- A fair ± 1 random walk. The state space is $\mathbb{Z}$, the transition probabilities are $p_{ij} = 1/2$ if $|i-j| = 1$, 0 otherwise. This is an example of a Markov chain that is also a martingale.

- A fair ± 1 random walk on a cycle. As above, but now the state space is $\mathbb{Z}/m$, the integers mod m. An example of a finite Markov chain.

- Random walks with absorbing/reflecting barriers.

- Random walk on a graph $G = (V, E)$. The state space is V, the transition probabilities are $p_{uv} = 1/d(u)$ if $uv \in E$.

  One can also have more general transition probabilities, where the probability of traversing a particular edge is a property of the edge and not the degree of its source. In principle we can represent any

Markov chain as a random walk on graph in this way: the states become vertices, and the transitions become edges, each labeled with its transition probability. It's conventional in this representation to exclude edges with probability 0 and include self-loops for any transitions $i \to i$.

If the resulting graph is small enough or has a nice structure, this can be a convenient way to draw a Markov chain.

- The Markov chain given by $X_{t+1} = X_t + 1$ with probability 1/2, and 0 with probability 1/2. The state space is $\mathbb{N}$.

- A finite-state machine running on a random input. The sequence of states acts as a Markov chain, assuming each input symbol is independent of the rest.

- A classic randomized algorithm for 2-SAT, due to Papadimitriou [Pap91]. State is a truth-assignment. The transitional probabilities are messy but arise from the following process: pick an unsatisfied clause, pick one of its two variables uniformly at random, and invert it. Then there is an absorbing state at any satisfying assignment. With a bit of work, it can be shown that the Hamming distance between the current assignment and some satisfying assignment follows a random walk biased toward 0, giving a satisfying assignment after $O(n^2)$ steps on average.

- A similar process works for 2-colorability, 3-SAT, 3-colorability, etc., although for **NP**-hard problems, it may take a while to reach an absorbing state. The constructive Lovász Local Lemma proof from §11.3.5 also follows this pattern.

9.1.2 Classification of states

Given a Markov chain, define

$$f_{ij}(n) = \Pr\left[j \notin \{X_1 \dots X_{n-1}\} \wedge j = X_n \mid X_0 = i\right]. \tag{9.1.2}$$

Then $f_{ij}(n)$ is the probability that the **first passage time** from i to j equals n. Define $f_{ij} = \sum_{n=1}^{\infty} f_{ij}(n)$; this is the probability that the chain ever reaches j starting from i. A state is **persistent** if $f_{ii} = 1$ and **transient** if $f_{ii} < 1$.

Finally, define the **mean recurrence time** $\mu_i = \sum_{n=1}^{\infty} n f_{ii}(n)$ if i is persistent and $\mu_i = \infty$ if i is transient. We use the mean recurrence time to

Parameter	Name	Definition
$p_{ij}(n)$	n-step transition probability	$\Pr[X_{t+n} = j \mid X_t = i]$
p_{ij}	Transition probability	$\Pr[X_{t+1} = j \mid X_t = i] = p_{ij}(1)$
$f_{ij}(n)$	First passage time	$\Pr[j \notin \{X_1 \dots X_n - 1\} \wedge X_n = j \mid X_0 = i]$
f_{ij}	Probability of reaching j from i	$\sum_{n=1}^{\infty} f_{ij}(n)$
μ_i	Mean recurrence time	$\mu_i = \sum_{n=1}^{\infty} n f_{ii}(n)$
π_i	Stationary distribution	$\pi_j = \sum_i \pi_i p_{ij} = 1/\mu_j$

Table 9.1: Markov chain parameters

f_{ii}	μ_i	classification
< 1	$= \infty$	transient
$= 1$	$= \infty$	null persistent
$= 1$	$< \infty$	non-null persistent

Table 9.2: Classification of Markov chain states

further classify persistent states. If the sum diverges, we say that the state is **null persistent**; otherwise it is **non-null persistent**.

These definitions are summarized in Tables 9.1 and 9.2.

The distinctions between different classes of states are mostly important in infinite Markov chains. In an irreducible finite Markov chain, all states are non-null persistent. To show this, pick some state i, and for each j let n_{ji} be the least n for which $p_{ji}(n)$ is nonzero. (This notation is not standard and will not be used after this paragraph.) Then from any starting state, if we run for at least $n_{\max} = \max_j n_{ji}$ steps, we have a probability of at least $p_{\min} = \min_j p_{ji}(n_{ji})$ of reaching i. This means that the expected time to reach i starting from any state (including i) is at most $n_{\max}/p_{\min} < \infty$, giving $\mu_i < \infty$. More generally, any state in a finite Markov chain is either transient or non-null persistent, and there is always at least one non-null persistent state.

In contrast, even in an irreducible infinite chain we may have null persistent or even transient states. For example, in a random ± 1 walk on $\mathbb{Z}$, all states are null persistent, because once I am at $+1$ the expected time to get back to 0 is unbounded; see the analysis in §8.2 for a proof. In a not-so-random walk on $\mathbb{Z}$ where $X_{t+1} = X_t + 1$ always, all states are transient: once I leave x for $x + 1$, I'm never coming back. For a more exotic example, in a random walk on $\mathbb{Z}^3$ (where at each step we have a 1/6 chance of moving ± 1 in one of the three coordinates), all states are transient. Intuitively, this is because after n steps, the particle is at distance $\Theta(\sqrt{n})$ on average from its

origin, and since there are $\Theta(n^{3/2})$ points in this range but only n that are visited by the random walk, the chance that any particular point is visited goes to zero in the limit (the actual proof requires a better argument than this).

The **period** $d(i)$ of a state i is $\gcd(\{n \mid p_{ii}(n) > 0\})$. A state i is **aperiodic** if $d(i) = 1$, and **periodic** otherwise. A Markov chain is aperiodic if all its states are aperiodic.

A useful property of an aperiodic state i is that there is always some minimum $n_{\min}$ such that $p_{ii}(n) > 0$ for all $n \geq n_{\min}$. This is a consequence of the solution to the **Frobenius problem**, which says that for any sequence of values $m_1, \ldots m_k$ with $\gcd(m_1, \ldots, m_k) = 1$, every sufficiently large n can be written as $\sum a_i m_i$ for non-negative integer a_i. We won't attempt to prove this result here, but will use it later.

The most well-behaved states are the aperiodic non-null persistent states; these are called **ergodic**. A Markov chain is ergodic if all its states are.

9.1.3 Reachability

State i **communicates** with state j if $p_{ij}(n) > 0$ for some n; i.e., it is possible to reach j from i. This is often abbreviated as $i \to j$. Two states i and j **intercommunicate** if $i \to j$ and $j \to i$; this is similarly abbreviated as $i \leftrightarrow j$. If $i \leftrightarrow j$, it's not hard to show that i and j have the same period and classification. A set of states S is **closed** if $p_{ij} = 0$ for all $i \in S$, $j \notin S$, and **irreducible** if $i \leftrightarrow j$ for all i, j in S.

Using graph-theoretic terminology, $i \to j$ if j is **reachable** from i (through edges corresponding to transitions with nonzero probabilities), and a set of states is closed if it is **strongly connected**. We can thus decompose a Markov chain into strongly-connected components, and observe that all states in a particular strongly-connected component are persistent if and only if the strongly-connected component has no outgoing edges (i.e., is closed). For finite chains there must be at least one closed strongly-connected component (corresponding to a sink in the quotient DAG); this gives an instant proof that finite chains have persistent states.

If the entire chain forms a single strongly-connected component, we say it is **irreducible**.

9.2 Stationary distributions

The key useful fact about irreducible Markov chains is that (at least in the finite case) they have **stationary distributions**, where a stationary

distribution is a distribution π (non-negative row vector summing to 1) such that $\pi P = \pi$. The full theorem is:

Theorem 9.2.1. *An irreducible Markov chain has a stationary distribution π if and only if all states are non-null persistent, and if π exists, then $\pi_i = 1/\mu_i$ for all i.*

The proof of this theorem takes a while, so we won't do it (see [GS01, §6.4] if you want to see the details). For finite chains, the first part follows from the fact that the all-ones vectors is a right eigenvector for P with eigenvalue 1 (this is a fancy way of saying $P\mathbf{1} = \mathbf{1}$), which means that there is some left eigenvector for P that also has eigenvalue 1 ($\pi P = \pi$); we can also appeal to the **Perron-Frobenius theorem**,[2] which says this and more about the eigenvalues of stochastic matrices. The second part ($\pi_i = 1/\mu_i$) is just a special case of the **renewal theorem**.[3] The intuition is that if I get back to i once every μ_i steps on average, then I should expect to be spending $1/\mu_i$ of my time there in the limit.

For reducible Markov chains, there is a stationary distribution on each closed irreducible subset, and the stationary distributions for the chain as a whole are all convex combinations of these stationary distributions.

Examples: In the random walk on $\mathbb{Z}_m$ the stationary distribution satisfies $\pi_i = 1/m$ for all i (immediate from symmetry). By contrast, the random walk on $\mathbb{Z}$ has no stationary distribution (the states are all null persistent). The process on $\mathbb{N}$ where $p_{ij} = 1/2$ for $j = i + 1$ or $j = 0$ has a stationary distribution π given by $\pi_i = 2^{-i-1}$; this is an example of an infinite chain that nonetheless has only non-null persistent states. For a random walk with absorbing barriers at $\pm n$, there is no unique stationary distribution; instead, any vector π where $\pi_i = 0$ unless $i = \pm n$ and $\pi 1 = 1$ is stationary.

It's generally not a good strategy to compute π by computing μ first. Instead, solve the matrix equation $\pi P = \pi$ by rewriting it as $\pi(P - I) = 0$ and adding the constraint $\pi 1 = 1$. You can then compute $\mu_i = 1/p_i$ if you still need μ_i for some reason.

9.2.1 The ergodic theorem

The basic version of the **ergodic theorem** says that if an aperiodic irreducible Markov chain has a stationary distribution π, then it converges to π if we run it long enough. (This also implies that π is unique.)

[2] `http://en.wikipedia.org/wiki/Perron-Frobenius_theorem`.

[3] `http://en.wikipedia.org/wiki/Renewal_theory`.

This is not terribly hard to prove for finite chains, using a very clever technique called **coupling**, where we take two copies of the chain, once of which starts in an arbitrary distribution, and one of which starts in the stationary distribution, and show that we can force them to converge to each other by carefully correlating their transitions. Since coupling is a generally useful technique for bounding rate of convergence, we'll use it to give the proof of the ergodic theorem below.

A more general version of the ergodic theorem says that for any Markov chain and any aperiodic state j, then $p_{jj}(n) \to 1/\mu_j$ and $p_{ij}(n) \to f_{ij}/\mu_j$. This allows us in principle to compute the limit distribution for any aperiodic Markov chain with a given starting distribution x by summing $\sum_i x_i f_{ij}/\mu_j$. It also implies that transient and null persistent states vanish in the limit.

9.2.1.1 Proof for finite chains

We'll do this by constructing a **coupling** between two copies of the Markov chain, which as stated before is a joint distribution on two processes that tends to bring them together while looking like the original process when restricted to one sub-process or the other. In the first process $\{X_t\}$, we start with an arbitrary initial distribution for X_0. In the second $\{Y_t\}$, we start with $\Pr[Y_t = i] = \pi_i$ for all i. We now evolve the joint process $\{(X_t, Y_t)\}$ by the rule $(i, i') \to (j, j')$ with probability $p_{ij}p_{i'j'}$ if $i \neq i'$ and $(i, i') \to (j, j')$ with probability p_{ij} if $i = i'$ and $j = j'$ and 0 if $i = i'$ and $j \neq j'$. In other words, we let X_t and Y_t both wander around independently until they collide, after which they stick together and wander around together.

The reason this shows convergence is that we can write

$$\begin{aligned}\Pr[X_t = j] &= \Pr[X_t = Y_t]\Pr[Y_t = j \mid X_t = Y_t] \\ &\quad + \Pr[X_t \neq Y_t]\Pr[X_t = j \mid X_t \neq Y_t]\end{aligned} \tag{9.2.1}$$

and similarly

$$\begin{aligned}\Pr[Y_t = j] &= \Pr[X_t = Y_t]\Pr[Y_t = j \mid X_t = Y_t] \\ &\quad + \Pr[X_t \neq Y_t]\Pr[Y_t = j \mid X_t \neq Y_t]\,.\end{aligned} \tag{9.2.2}$$

The sneaky bit here is substituting $\Pr[Y_t = j \mid X_t = Y_t]$ for $\Pr[X_t = j \mid X_t = Y_t]$ in (9.2.1), giving us a connection between $\Pr[X_t = j]$ and $\Pr[Y_t = j]$ that we will exploit by showing $\Pr[X_t \neq Y_t]$ goes to 0 in the limit as t goes to infinity.

Subtract (9.2.2) from (9.2.1) to get

$$\Pr[X_t = j] - \Pr[Y_t = j] = \\ \Pr[X_t \neq Y_t] \left(\Pr[X_t = j \mid X_t \neq Y_t] - \Pr[Y_t = j \mid X_t \neq Y_t]\right). \quad (9.2.3)$$

If we can show $\Pr[X_t \neq Y_t]$ goes to zero, then the right-hand side of (9.2.3) also goes to zero, implying that the difference between $\Pr[X_t = j]$ and $\Pr[Y_t = j]$ goes to zero as well. But $\Pr[Y_t = j]$ is just π_j, so this shows that $\Pr[X_t = j]$ converges to π_j as claimed.

So now we just need to show $\Pr[X_t \neq Y_t] \to 0$. This is equivalent to showing that $\Pr[X_t = Y_t] \to 1$, which is easier to reason about, since we just need to show that X_t and Y_t eventually collide with probability 1.

Because each state in the chain is aperiodic, we have that $p_{ii}(n) > 0$ for any i and $n \geq n_{\min}$ for some $n_{\min}$. Because it is finite, we can use the same $n_{\min}$ for all states, by taking the max over all the values for individual states. Because the chain is irreducible, we also have that $p_{ij}(m_{ij}) > 0$ for each i and j and some m_{ij}.

Let $N = \max_{ij} m_{ij} + n_{\min}$. Then for any i and j, we have $p_{ij}(N) \geq p_{ij}(m_{ij}) \cdot p_{jj}(N - m_{ij}) > 0$. So we can get from any state to any other state in N steps with nonzero probability. Let p be the minimum such probability over all i and j.

Suppose now that we start with $X_0 = i$, $Y_0 = j$. Then $p_{ii}(N)$ and $p_{ij}(N)$ are both at least p, so if we run the chains independently, after N steps there is at least a p^2 chance that $X_N = Y_N = i$. If this fails, then it is still the case that $p_{X_N,i} \geq p$ and $p_{Y_N,i} \geq p$, so after an additional N states, there is at least a p^2 chance that $X_{2N} = Y_{2N} = i$. Continuing in this way we have a probability of at most $(1 - p^2)^k$ that the two processes have not collided after kN. This goes to zero in the limit, proving the theorem.

9.3 Reversible chains

Some chains have the property of being **reversible**. Formally, a chain with transition probabilities p_{ij} is reversible if there is a distribution π such that

$$\pi_i p_{ij} = \pi_j p_{ji} \quad (9.3.1)$$

for all i, j. These are called the **detailed balance equations**—they say that in the stationary distribution, the probability of seeing a transition from i to j is equal to the probability of seeing a transition from j to i).

If this is the case, then $\sum_i \pi_i p_{ij} = \sum_i \pi_j p_{ji} = \pi_j$, which means that π is stationary.

This often gives a very quick way to compute the stationary distribution, since if we know π_i, and $p_{ij} \neq 0$, then $\pi_j = \pi_i p_{ij}/p_{ji}$. If the transition probabilities are reasonably well-behaved (for example, if $p_{ij} = p_{ij}$ for all i, j), we may even be able to characterize the stationary distribution up to a constant multiple even if we have no way to efficiently enumerate all the states of the process.

The reason that such a chain is called reversible is that if we start in the stationary distribution at time 0, then the sequence of random variables $(X_0, \ldots, X_t)$ has *exactly the same distribution* as the reversed sequence $(X_t, \ldots, X_0)$. [4]

Note that a reversible chain can't have a period higher than 2, since we can always step back to where we came from.

9.3.1 Examples

Random walk on a graph Given two adjacent vertices u and v, we have $\pi_v = \pi_u d(v)/d(u)$. This suggests $\pi_v = cd(v)$ for some constant c and all v (the stationary distribution is proportional to the degree), and the constant c is determined by $\sum_v \pi_v = c \sum d(v) = 1$, giving $\pi_v = d(v)/\sum_u d(u)$.

Random walk on a weighted graph Here each edge has a weight w_{uv} where $0 < w_{uv} = w_{vu} < \infty$, with self-loops permitted. A step of the random walk goes from u to v with probability $w_{uv}/\sum_{v'} w_{uv'}$. It is easy to show that this random walk has stationary distribution $\pi_u = \sum_u w_{uv}/\sum_u \sum_v w_{uv}$, generalizing the previous case, and that the resulting Markov chain satisfies the detailed balance equations.

Random walk with uniform stationary distribution Now let Δ be the maximum degree of the graph, and traverse each each with probability $1/\Delta$, staying put on each vertex u with probability $1 - d(u)/\Delta$. The stationary distribution is uniform, since for each pair of vertices u and v we have $p_{uv} = p_{vu} = 1/\Delta$ if u and v are adjacent and 0 otherwise.

[4] Proof: Start with $\Pr[\forall i X_i = x_i] = \pi_{x_0} \prod_{i=0}^{t} p_{x_i x_{i+1}}$. Now observe that we can move the π_{x_0} across the first factor to get $p_{x_1 x_0} \pi_{x_1} \prod_{i=1}^{t} p_{x_i x_{i+1}}$ and in general $\left(\prod_{i=0}^{j-1} p_{x_{i+1} x_i}\right) \pi_{x_j} \left(\prod_{i=j}^{t} p_{x_i x_{i+1}}\right)$. At $j = t$ we get $\pi_{x_t} \prod_{i=0}^{t-1} p_{x_{i+1} x_i} = \Pr[\forall i X_i = x_{t-i}]$.

9.3.2 Time-reversed chains

Another way to get a reversible chain is to take an arbitrary chain with a stationary distribution and rearrange it so that it can run both forwards and backwards in time.

Given a finite Markov chain with transition matrix P and stationary distribution π, define the corresponding **time-reversed chain** with matrix P^* where $\pi_i p_{ij} = \pi_j p^*_{ji}$.

To make sure that this actually works, we need to verify that:

1. The matrix P^* is stochastic:

$$\begin{aligned}\sum_j p^*_{ij} &= \sum_j p_{ji}\pi_j/\pi_i \\ &= \pi_i/\pi_i \\ &= 1.\end{aligned}$$

2. The reversed chain has the same stationary distribution as the original chain:

$$\begin{aligned}\sum_j \pi_j p^*_{ji} &= \sum_j \pi_i p_{ij} \\ &= \pi_i.\end{aligned}$$

3. And that in general P^*'s paths starting from the stationary distribution are a reverse of P's paths starting from the same distribution. For length-1 paths, this is just $\pi_j p^*_{ji} = \pi_i p_{ij}$. For longer paths, this follows from an argument similar to that for reversible chains.

This gives an alternate definition of a reversible chain as a chain for which $P = P^*$.

We can also use time-reversal to generate reversible chains from arbitrary chains. The chain with transition matrix $(P + P^*)/2$ (corresponding to moving 1 step forward or back with equal probability at each step) is always a reversible chain.

Examples:

- Given a biased random walk on a cycle that moves right with probability p and left with probability q, its time-reversal is the walk that moves left with probability p and right with probability q. (Here the fact that the stationary distribution is uniform makes things simple.) The average of this chain with its time-reversal is an unbiased random walk.

- Given the random walk defined by $X_{t+1} = X_t + 1$ with probability 1/2 and 0 with probability 1/2, we have $\pi_i = 2^{-i-1}$. This is not reversible (there is a transition from 1 to 2 but none from 2 to 1), but we can reverse it by setting $p^*_{ij} = 1$ for $i = j + 1$ and $p^*_{0i} = 2^{-i-1}$. (Check: $\pi_i p_{ii+1} = 2^{-i-1}(1/2) = \pi_{i+1} p^*_{i+1i} = 2^{-i-2}(1); \pi_i p_{i0} = 2^{-i-1}(1/2) = \pi_0 p^*_{0i} = (1/2)2^{-i-1}$.)

Reversed versions of chains with messier stationary distributions are messier.

9.3.3 The Metropolis-Hastings algorithm

The basic idea of the **Metropolis-Hastings algorithm**[MRR+53, Has70] (sometimes just called **Metropolis**) is that we start with a reversible Markov chain P with a known stationary distribution π, but we'd rather get a chain Q on the same states with a different stationary distribution μ, where $\mu_i = f(i)/\sum_j f(j)$ is proportional to some function $f \geq 0$ on states that we can compute easily.

A typical application is that we want to sample according to $\Pr[i \mid A]$, but A is highly improbable (so we can't just use **rejection sampling**, where we sample random points from the original distribution until we find one for which A holds), and $\Pr[i \mid A]$ is easy to compute for any fixed i but tricky to compute for arbitrary events (so we can't use divide-and-conquer). If we let $f(i) \propto \Pr[i \mid A]$, then Metropolis-Hastings will do exactly what we want, assuming it converges in a reasonable amount of time.

Let q be the transition probability for Q. Define, for $i \neq j$,

$$\begin{aligned} q_{ij} &= p_{ij} \min\left(1, \frac{\pi_i f(j)}{\pi_j f(i)}\right) \\ &= p_{ij} \min\left(1, \frac{\pi_i \mu_j}{\pi_j \mu_i}\right) \end{aligned}$$

and let q_{ii} be whatever probability is left over. Now consider two states i and j, and suppose that $\pi_i f(j) \geq \pi_j f(i)$. Then

$$q_{ij} = p_{ij}$$

which gives

$$\mu_i q_{ij} = \mu_i p_{ij},$$

while

$$\begin{aligned}\mu_j q_{ji} &= \mu_j p_{ji}(\pi_j \mu_i / \pi_i \mu_j) \\ &= p_{ji}(\pi_j \mu_i / \pi_i) \\ &= \mu_i (p_{ji} \pi_j)/\pi_i \\ &= \mu_i (p_{ij} \pi_i)/\pi_i \\ &= \mu_i p_{ij}\end{aligned}$$

(note the use of reversibility of P in the second-to-last step). So we have $\mu_j q_{ji} = \mu_i p_{ij} = \mu_i q_{ij}$ and Q is a reversible Markov chain with stationary distribution μ.

We can simplify this when our underlying chain P has a uniform stationary distribution (for example, when it's the random walk on a graph with maximum degree Δ, where we traverse each edge with probability $1/\Delta$). Then we have $\pi_i = \pi_j$ for all i, j, so the new transition probabilities q_{ij} are just $\frac{1}{\Delta}\min(1, f(j)/f(i))$. Most of our examples of reversible chains will be instances of this case (see also [MU05, §10.4.1]).

9.4 The coupling method

In order to use the stationary distribution of a Markov chain to do sampling, we need to have a bound on the rate of convergence to tell us when it is safe to take a sample. There are two standard techniques for doing this: coupling, where we show that a copy of the process starting in an arbitrary state can be made to converge to a copy starting in the stationary distribution; and spectral methods, where we bound the rate of convergence by looking at the second-largest eigenvalue of the transition matrix. We'll start with coupling because it requires less development.

(See also [Gur00] for a survey of the relationship between the various methods.)

Note: these notes will be somewhat sketchy. If you want to read more about coupling, a good place to start might be Chapter 11 of [MU05]; Chapter 4-3 (`http://www.stat.berkeley.edu/~aldous/RWG/Chap4-3.pdf`) of the unpublished but nonetheless famous **Aldous-Fill manuscript** (`http://www.stat.berkeley.edu/~aldous/RWG/book.html`, [AF01]), which is a good place to learn about Markov chains and Markov chain Monte Carlo methods in general; or even an entire book [Lin92]. We'll mostly be using examples from the Aldous-Fill text.

9.4.1 Total variation distance

In order to show that we are converging, we need some measure of how close two distributions on states are. The natural measure for coupling arguments is the **total variation distance** , defined below.

Definition 9.4.1. *Let X and Y be random variables defined on the same probability space. Then the* ***total variation distance*** *between X and Y, written $d_{TV}(X,Y)$ is given by*

$$d_{TV}(X,Y) = \max_A \left(\Pr[X \in A] - \Pr[Y \in A]\right), \tag{9.4.1}$$

where the maximum is taken over all sets A for which $\Pr[X \in A]$ and $\Pr[Y \in A]$ are both defined.[5]

An equivalent definition is

$$d_{TV}(X,Y) = \max_A |\Pr[X \in A] - \Pr[Y \in A]|.$$

The reason this is equivalent is that if $\Pr[X \in A] - \Pr[Y \in A]$ is negative, we can replaced A by its complement.

Less formally, given any test set A, X and Y land in A with probabilities that differ by at most $d_{TV}(X,Y)$. This is usually what we want for sampling, since this says that if we are testing some property (represented by A) of the states we are sampling, the answer we get for how likely this property is to occur is close to the correct answer.

9.4.2 The coupling lemma

So now we want to use a coupling to prove convergence in total variation distance. The idea is to show that, for random variables X_t, Y_t corresponding to states of the coupled processes, $d_{TV}(X_t, Y_t) \leq \Pr[X_t \neq Y_t]$, and then show that this probability is low when t is large.

The tool that makes this work is the **Coupling Lemma**:

Lemma 9.4.2. *For any discrete random variables X and Y,*

$$d_{TV}(X,Y) \leq \Pr[X \neq Y].$$

[5] For discrete random variables, this just means all A, since we can write $\Pr[X \in A]$ as $\sum_{x \in A} \Pr[X = x]$. For continuous random variables, we want that $X^{-1}(A)$ and $Y^{-1}(A)$ are both measurable. If our X and Y range over the states of a countable Markov chain, we will be working with discrete random variables, so we can just consider all A. Another complication with continuous random variables is that we may need to change (9.4.1) to use sup instead of max.

Proof. Let A be any set for which $\Pr[X \in A]$ and $\Pr[Y \in A]$ are defined. Then

$$\begin{aligned}\Pr[X \in A] &= \Pr[X \in A \wedge Y \in A] + \Pr[X \in A \wedge Y \notin A],\\ \Pr[Y \in A] &= \Pr[X \in A \wedge Y \in A] + \Pr[X \notin A \wedge Y \in A],\end{aligned}$$

and thus

$$\begin{aligned}\Pr[X \in A] - \Pr[Y \in A] &= \Pr[X \in A \wedge Y \notin A] - \Pr[X \notin A \wedge Y \in A]\\ &\leq \Pr[X \in A \wedge Y \notin A]\\ &\leq \Pr[X \neq Y].\end{aligned}$$

Since this holds for any particular set A, it also holds when we take the maximum over all A to get $d_{TV}(X, Y)$. □

Armed with this bound, we can now go hunting for couplings that make $X_t = Y_t$ with high probability for large enough t. Designing such a coupling will depend on the structure of our Markov chain. Examples are given in the following sections.[6]

9.4.3 Random walk on a cycle

Let's suppose we do a random walk on $\mathbb{Z}_m$, where to avoid periodicity at each step we stay put with probability 1/2, move counterclockwise with probability 1/4, and move clockwise with probability 1/4: in other words, we are doing a lazy unbiased random walk. What's a good choice for a coupling to show this process converges quickly?

Specifically, we need to create a joint process (X_t, Y_t), where each of the marginal processes X and Y looks like a lazy random walk on $\mathbb{Z}_m$, X_0 has whatever distribution our real process starts with, and Y_0 has the stationary distribution. Our goal is to structure the combined process so that $X_t = Y_t$ as soon as possible.

[6] A curious fact: in principle there *always* exists a coupling between X and Y such that $d_{TV}(X_t, Y_t) = \Pr[X_t \neq Y_t]$. The intuition is that we can take trajectories in the X and Y processes that end in the same state and match them up as much as we can; done right, this makes the event $X_t = Y_t$ occur as often as possible given the mismatches between $\Pr[X_t = x]$ and $\Pr[Y_t = x]$. The only problem with this is that (a) we may have to know a lot about how the convergence comes about to know which trajectories to follow, because we are essentially picking the entire path for both processes at time 0; and (b) we may have to correlate the initial states X_0 and Y_0, which is easy if X_0 puts all its weight on one point, but gets complicated if the distribution on X_0 is not trivial. So this fact is not particularly useful in practice.

Let $Z_t = X_t - Y_t \pmod m$. If $Z_t = 0$, then X_t and Y_t have collided and we will move both together. If $Z_t \neq 0$, then flip a coin to decided whether to move X_t or Y_t; whichever one moves then moves up or down with equal probability. It's not hard to see that this gives a probability of exactly $1/2$ that $X_{t+1} = X_t$, $1/4$ that $X_{t+1} = X_t + 1$, and $1/4$ that $X_{t+1} = X_t - 1$, and similarly for Y_t. So the transition functions for X and Y individually are the same as for the original process.

Whichever way the first flip goes, we get $Z_{t+1} = Z_t \pm 1$ with equal probability. So Z acts as an unbiased random walk on $\mathbb{Z}_m$ with an absorbing barriers at 0; this is equivalent to a random walk on $0 \ldots m$ with absorbing barriers at both endpoints. The expected time for this random walk to reach a barrier starting from an arbitrary initial state is at most $m^2/4$, so if τ is the first time at which $X_\tau = Y_\tau$, we have $\mathrm{E}[\tau] \leq m^2/4$.[7]

Using Markov's inequality, after $t = 2(m^2/4) = m^2/2$ steps we have $\Pr[X_t \neq Y_t] = \Pr[\tau > m^2/2] \leq \frac{\mathrm{E}[\tau]}{m^2/2} \leq 1/2$. We can also iterate the whole argument, starting over in whatever state we are in at time t if we don't converge. This gives at most a $1/2$ chance of not converging for each interval of $m^2/2$ steps. So after $\alpha m^2/2$ steps we will have $\Pr[X_t \neq Y_t] \leq 2^{-\alpha}$. This gives a **mixing time** to reach $d_{TV} \leq \epsilon$ less than $\frac{1}{2}m^2 \lg(2/\epsilon)$.

The choice of 2 for the constant in Markov's inequality could be improved. The following lemma gives an optimized version of this argument:

Lemma 9.4.3. *Let the expected* ***coupling time****, at which two coupled processes $\{X_t\}$ and $\{Y_t\}$ starting from an arbitrary state are first equal, be T. Then $d_{TV}(X_{T_\epsilon}, Y_{T_\epsilon}) \leq \epsilon$ for $T_\epsilon \geq Te\ln(2/\epsilon)$.*

Proof. Essentially the same argument as above, but replacing 2 with a constant c to be determined. Suppose we restart the process every cT steps. Then at time t we have a total variation bounded by $2c^{-\lfloor t/cT \rfloor}$. The expression $c^{-t/cT}$ is minimized by minimizing $c^{-1/c}$ or equivalently $-\ln c/c$, which occurs at $c = e$. This gives a time to reach an $\epsilon/2$ probability of of $Te\ln(2/\epsilon)$. □

It's worth noting that the random walk example was very carefully rigged to make the coupling argument clean. A similar argument still works (perhaps with a change in the bound) for other ergodic walks on the ring, but the details are messier.

[7] If we know that Y_0 is uniform, then Z_0 is also uniform, and we can use this fact to get a slightly smaller bound on $\mathrm{E}[\tau]$, around $m^2/6$. But this will cause problems if we want to re-run the coupling starting from a state where X_t and Y_t have not yet converged.

9.4.4 Random walk on a hypercube

Start with a bit-vector of length n. At each step, choose an index uniformly at random, and set the value of the bit-vector at that index to 0 or 1 with equal probability. How long until we get a nearly-uniform distribution over all 2^n possible bit-vectors?

Here we apply the same transformation to both the X and Y vectors. It's easy to see that the two vectors will be equal once every index has been selected once. The waiting time for this to occur is just the waiting time nH_n for the coupon collector problem. We can either use this expected time directly to show that the process mixes in time $O(n \log n \log(1/\epsilon))$ as above, or we can use known sharp concentration bounds on coupon collector (see [MU05, §5.4.1] or [MR95, §3.6.3]), which shows $\lim_{n\to\infty} \Pr[T \geq n(\ln n + c)] = 1 - \exp(-\exp(-c)))$ to show that in the limit $n \ln n + n \ln\ln(1/(1-\epsilon)) = n \ln n + O(n \log(1/\epsilon))$ is enough.[8]

We can improve the bound slightly by observing that, on average, half the bits in X_0 and Y_0 are already equal; doing this right involves summing over a lot of cases, so we won't do it.

This is an example of a Markov chain with the **rapid mixing** property: the mixing time is polylogarithmic in the number of states (2^n in this case) and $1/\epsilon$. For comparison, the random walk on the ring is not rapid mixing, because the coupling time is polynomial in $n = m$ rather than $\log n$.

9.4.5 Various shuffling algorithms

Here we have a deck of n cards, and we repeatedly apply some random transformation to the deck to converge to a stationary distribution that is uniform over all permutations of the cards (usually this is obvious by symmetry, so we won't bother proving it). Our goal is to show that the expected **coupling time** at which our deck ends up in the same permutation as an initially-stationary deck is small. We do this by counting how many cards S_t are in the same position in both decks, and showing that, for a suitable coupling, (a) this quantity never decreases, and (b) it increases with some nonzero probability at each step. The expected coupling time is then $\sum_k 1/\Pr[S_{t+1} = k+1 \mid S_t = k]$.

[8] This is a little tricky; we don't know from this bound alone how fast the probability converges as a function of n, so to do this right we need to look into the bound in more detail.

9.4.5.1 Move-to-top

This is a variant of card shuffling that is interesting mostly because it gives about the easiest possible coupling argument. At each step, we choose one of the cards uniformly at random (including the top card) and move it to the top of the deck. How long until the deck is fully shuffled, i.e., until the total variation distance between the actual distribution and the stationary distribution is bounded by ϵ?

Here the trick is that when we choose a card to move to the top in the X process, we choose the same card in the Y process. It's not hard to see that this links the two cards together so that they are always in the same position in the deck in all future states. So to keep track of how well the coupling is working, we just keep track of how many cards are linked in this way, and observe that as soon as $n-1$ are, the two decks are identical.

Note: Unlike some of the examples below, we don't consider two cards to be linked just because they are in the same position. We are only considering cards that have gone through the top position in the deck (which corresponds to some initial segment of the deck, viewed from above). The reason is that these cards never become unlinked: if we pick two cards from the initial segment, the cards above them move down together. But deeper cards that happen to match might become separated if we pull a card from one deck that is above the matched pair while its counterpart in the other deck is below the matched pair.

Having carefully processed the above note, given k linked cards the probability that the next step links another pair of cards is exactly $(n-k)/n$. So the expected time until we get $k+1$ cards is $n/(n-k)$, and if we sum these waiting times for $k = 0 \ldots n-1$, we get nH_n, the waiting time for the coupon collector problem. So the bound on the mixing time is the same as for the random walk on a hypercube.

9.4.5.2 Random exchange of arbitrary cards

Here we pick two cards uniformly and independently at random and swap them. (Note there is a $1/n$ chance they are the same card; if we exclude this case, the Markov chain has period 2.) To get a coupling, we reformulate this process as picking a random card and a random location, and swapping the chosen card with whatever is in the chosen location in both the X and Y processes.

First let's observe that the number of linked cards never decreases. Let x_i, y_i be the position of card i in each process, and suppose $x_i = y_i$. If neither

card i nor position x_i is picked, i doesn't move, and so it stays linked. If card i is picked, then both copies are moved to the same location; it stays linked. If position x_i is picked, then it may be that i becomes unlinked; but this only happens if the card j that is picked has $x_j \neq y_j$. In this case j becomes linked, and the number of linked cards doesn't drop.

Now we need to know how likely it is that we go from k to $k+1$ linked cards. We've already seen a case where the number of linked cards increases; we pick two cards that aren't linked and a location that contains cards that aren't linked. The probability of doing this is $((n-k)/n)^2$, so our total expected waiting time is $n^2 \sum (n-k)^{-2} = n^2 \sum k^{-2} \leq n\pi^2/6$ (see `http://en.wikipedia.org/wiki/Basel_problem` for an extensive discussion of the useful but non-obvious fact that $\sum_{k \in \mathbb{N}_+} k^{-2} = \zeta(2) = \pi^2/6$.) The final bound is $O(n^2 \log(1/\epsilon))$.

This bound is much worse that the bound for move-to-top, which is surprising. In fact, the real bound is $O(n \log n)$ with high probability, although the proof uses very different methods (see `http://www.stat.berkeley.edu/~aldous/RWG/Chap7.pdf`). This shows that the coupling method doesn't always give tight bounds (or perhaps we need a better coupling?).

9.4.5.3 Random exchange of adjacent cards

Suppose now that we only swap adjacent cards. Specifically, we choose one of the n positions i in the deck uniformly at random, and then swap the cards a positions i and $i+1 \pmod n$ with probability $1/2$. (The $1/2$ is there for the usual reason of avoiding periodicity.)

So now we want a coupling between the X and Y processes where each possible swap occurs with probability $\frac{1}{2n}$ on both sides, but somehow we correlate things so that like cards are pushed together but never pulled apart. The trick is that we will use the same position i on both sides, but be sneaky about when we swap. In particular, we will aim to arrange things so that once some card is in the same position in both decks, both copies move together, but otherwise one copy changes its position by ± 1 relative to the other with a fixed probability $\frac{1}{2n}$.

The coupled process works like this. Let D be the set of indices i where the same card appears in both decks at position i or at position $i+1$. Then we do:

1. For $i \in D$, swap $(i, i+1)$ in both decks with probability $\frac{1}{2n}$.
2. For $i \notin D$, swap $(i, i+1)$ in the X deck only with probability $\frac{1}{2n}$.

3. For $i \notin D$, swap $(i, i+1)$ in the Y deck only with probability $\frac{1}{2n}$.

4. Do nothing with probability $\frac{|D|}{2n}$.

It's worth checking that the total probability of all these events is $|D|/2n + 2(n - |D|)/2n + |D|/2n = 1$. More important is that if we consider only one of the decks, the probability of doing a swap at $(i, i+1)$ is exactly $\frac{1}{2n}$ (since we catch either case 1 or 2 for the X deck or 1 or 3 for the Y deck).

Now suppose that some card c is at position x in X and y in Y. If $x = y$, then both x and $x - 1$ are in D, so the only way the card can move is if it moves in both decks: linked cards stay linked. If $x \neq y$, then c moves in deck X or deck Y, but not both. (The only way it can move in both is in case 1, where $i = x$ and $i + 1 = y$ or vice verse; but in this case i can't be in D since the copy of c at position x doesn't match whatever is in deck Y, and the copy at position y doesn't match what's in deck X.) In this case the distance $x - y$ goes up or down by 1 with equal probability $1/2n$. Considering $x - y \pmod{n}$, we have a "lazy" random walk that moves with probability $1/n$, with absorbing barriers at 0 and n. The worst-case expected time to converge is $n(n/2)^2 = n^3/4$, giving $\Pr\left[\text{time for } c \text{ to become linked} \geq \alpha n^3/8\right] \leq 2^{-\alpha}$ using the usual argument. Now apply the union bound to get $\Pr\left[\text{time for every } c \text{ to become linked} \geq \alpha n^3/8\right] \leq n2^{-\alpha}$ to get an expected coupling time of $O(n^3 \log n)$. In this case (say Aldous and Fill, quoting a result of David Bruce Wilson [Wil04]) the bound is optimal up to a constant factor.

9.4.5.4 Real-world shuffling

In real life, the handful of people who still use physical playing cards tend to use a **dovetail shuffle**, which is closely approximated by the reverse of a process where each card in a deck is independently assigned to a left or right pile and the left pile is place on top of the right pile. Coupling doesn't really help much here. Instead, the process can be analyzed using more sophisticated techniques due to Bayer and Diaconis [BD92]. The short version of the result is that $\Theta(\log n)$ shuffles are needed to randomize a deck of size n.

9.4.6 Path coupling

If the states of our Markov process are the vertices of a graph, we may be able to construct a coupling by considering a path between two vertices and showing how to shorten this path on average at each step. This technique

is known as path coupling. Typically, the graph we use will be the graph of possible transitions of the underlying Markov chain (possibly after making all edges undirected), but any graph works as long as we can show that there is a coupling that makes adjacent vertices likely to coalesce.[9]

There are two ideas at work here. The first is that the expected distance $\mathrm{E}\left[d(X_t, Y_t)\right]$ between X_t and Y_t in the graph gives an upper bound on $\Pr\left[X_t \neq Y_t\right]$ (by Markov's inequality, since if $X_t \neq Y_t$ then $d(X_t, Y_t) \geq 1$). The second is that to show that $\mathrm{E}\left[d(X_{t+1}, Y_{t+1}) \mid \mathcal{F}_t\right] \leq \alpha \cdot d(X_t, Y_t)$ for some $\alpha < 1$, it is enough to show how to contract a single edge, that is, to show that $\mathrm{E}\left[d(X_{t+1}, Y_{t+1} \mid d(X_t, Y_t) = 1\right] \leq \alpha$. The reason is that if we have a coupling that contracts one edge, we can apply this inductively along each edge in the path to get a coupling between all the vertices in the path that still leaves each pair with expected distance at most α. The result for the whole path then follows from linearity of expectation.

Formally, instead of just looking at X_t and Y_t, consider a path of intermediate states $X_t = Z_{0,t} Z_{1,t} Z_{2,t} \ldots Z_{m,t} = Y_t$, where $d(Z_{i,t}, Z_{i+1,t}) = 1$ for each i (the vertices are adjacent in the graph). We now construct a coupling only for adjacent nodes that reduces their distance on average. The idea is that $d(X_t, Y_t) \leq \sum d(Z_{i,t}, Z_{i+1,t})$, so if the distance between each adjacent pair shrinks on average, so does the total length of the path.

The coupling on each edge gives a joint conditional probability

$$\Pr\left[Z_{i,t+1} = z'_i, Z_{i+1,t+1} = z'_{i+1} \mid Z_{i,t} = z_i, Z_{i+1,t} = z_{i+1}\right].$$

We can extract from this a conditional distribution on $Z_{i+1,t+1}$ given the other three variables:

$$\Pr\left[Z_{i+1,t+1} = z'_{i+1} \mid Z_{i,t+1} = z'_i, Z_{i,t} = z_i, Z_{i+1,t} = z_{i+1}\right].$$

Multiplying these conditional probabilities together lets us computed a joint distribution on X_{t+1}, Y_{t+1} conditioned on X_t, Y_t, which is the ordinary coupling we really want.

It's worth noting that the path is entirely notional, and we don't actually use it to construct an explicit coupling or even keep it around between steps of the coupled Markov chains. The only purpose of $Z_0, Z_1, \ldots, Z_m$ is to show that X and Y move closer together. Even though we could imagine that we are coalescing these nodes together to create a new path at each step (or throwing in a few extra nodes if some Z_i, Z_{i+1} move away from each other),

[9] Assuming Markov chain is reversible, this does create a constraint that adjacent vertices in whatever graph we are using are at most *two* steps away from each other in the chain.

we could also imagine that we start with a fresh path at each step and throw it away as soon as it has done its job.

9.4.6.1 Sampling graph colorings

For example, let's look at sampling k-colorings of a graph with maximum degree Δ. We will assume that $k \geq 2\Delta + 1$.[10] For smaller values of k, it's possible that the chain still converges, but our analysis will not show this.

Consider the following chain on proper k-colorings of a graph with maximum degree Δ. At each step, we choose one of the n nodes v, compute the set S of colors not found on any of v's neighbors, and recolor v with a color chosen uniformly from S (which may be its original color).

Suppose $p_{ij} \neq 0$. Then i and j differ in at most one place v, and so the set S of permitted colors for each process—those not found on v's neighbors—are the same. This gives $p_{ij} = p_{ji} = \frac{1}{n \cdot |S|}$, and the detailed balance equations (9.3.1) hold when π_i is constant for all i. So we have a reversible Markov chain with a uniform stationary distribution. Now we will apply a path coupling to show that we converge to this stationary distribution reasonably quickly when k is large enough.

We'll think of colorings as vectors. Given two colorings x and y, let $d(x, y)$ be the Hamming distance between them, which is the number of nodes u for which $x_u \neq y_u$. To show convergence, we will construct a coupling that shows that $d(X^t, Y^t)$ converges to 0 over time starting from arbitrary initial points X^0 and Y^0.

A complication is that it's not immediately evident that the length of the shortest path from X^t to Y^t in the transition graph of our Markov chain is $d(X^t, Y^t)$. The problem is that it may not be possible to transform X^t into Y^t one node at a time without producing improper colorings. With enough colors, we can explicitly construct a short path between X^t and Y^t that uses only proper colorings; but for this particular process it is easier to simply extend the Markov chain to allow improper colorings, and show that our coupling works anyway. This also allows us to start with an improper coloring for X^0 if we are particularly lazy. The stationary distribution is not affected, because if i is a proper coloring and j is an improper coloring that differs from i in exactly one place, we have $p_{ij} = 0$ and $p_{ji} \neq 0$, so the detailed balance equations hold with $\pi_j = 0$.

[10] An analysis based on a standard coupling for a different version of the Markov chain that works for the same bound on k is given in [MU05, §11.5]. Even more sophisticated results and a history of the problem can be found in [DGM02].

The natural coupling to consider given adjacent X^t and Y^t is to pick the same node and the same new color for both, provided we can do so. If we pick the one node v on which they differ, and choose a color that is not used by any neighbor (which will be the same for both copies of the process, since all the neighbors have the same colors), then we get $X^{t+1} = Y^{t+1}$; this event occurs with probability at least $1/n$. If we pick a node that is neither v nor adjacent to it, then the distance between X and Y doesn't change; either both get a new identical color or both don't.

Things get a little messier when we pick some node u adjacent to v, an event that occurs with probability at most Δ/n. Let c be the color of v in X^t, c' the color of u in Y^t, and T the set of colors that do not appear among the other neighbors of v. Let $\ell = |T| \geq k - (\Delta - 1)$.

Conditioning on choosing u to recolor, X^{t+1} picks a color uniformly from $T \setminus \{c\}$ and Y^{t+1} picks a color uniformly from $T \setminus \{c'\}$. We'd like these colors to be the same if possible, but these are not the same sets, and they aren't even necessarily the same size.

Conditioning on the choice of u, there are three cases:

1. Neither c nor c' are in T. Then X^{t+1} and Y^{t+1} are choosing a new color from the same set, and we can make both choose the same color: the distance between X and Y is unchanged.

2. Exactly one of c and c' is in T. Suppose that it's c. Then $|T \setminus \{c\}| = \ell - 1$ and $|T \setminus \{c'\}| = |T| = \ell$. Let X^{t+1} choose a new color c'' first. Then let $Y_u^{t+1} = c''$ with probability $\frac{\ell-1}{\ell}$ (this gives a probability of $\frac{1}{\ell}$ of picking each color in $T \setminus \{c\}$, which is what we want), and let $Y_u^{t+1} = c$ with probability $\frac{1}{\ell}$. Now the distance between X and Y increases with probability $\frac{1}{\ell}$.

3. Both c and c' are in T. For each c'' in $T \setminus \{c, c'\}$, let $X_u^{t+1} = Y_u^{t+1} = c''$ with probability $\frac{1}{\ell-1}$; since there are $\ell - 2$ such c'', this accounts for $\frac{\ell-2}{\ell-1}$ of the probability. Assign the remaining $\frac{1}{\ell-1}$ to $X_u^{t+1} = c', Y_u^{t+1} = c$. In this case the distance between X and Y increases with probability $\frac{1}{\ell-1}$, making this the worst case.

Putting everything together, we have a $1/n$ chance of picking a node that guarantees to reduce $d(X, Y)$ by 1, and at most a Δ/n chance of picking a node that may increase $d(X, Y)$ by at most $\frac{1}{\ell-1}$ on average, where $\ell \geq$

$k - \Delta + 1$, giving a maximum expected increase of $\frac{\Delta}{n} \cdot \frac{1}{k-\Delta}$. So

$$\begin{aligned}
\mathrm{E}\left[d(X^{t+1}, Y^{t+1}) - d(X^t, Y^t) \;\middle|\; d(X^t, Y^t) = 1\right] &\leq \frac{-1}{n} + \frac{\Delta}{n} \cdot \frac{1}{k-\Delta} \\
&= \frac{1}{n}\left(-1 + \frac{\Delta}{k-\Delta}\right) \\
&= \frac{1}{n}\left(\frac{-(k-\Delta)+\Delta}{k-\Delta}\right) \\
&= -\frac{1}{n}\left(\frac{k-2\Delta}{k-\Delta}\right).
\end{aligned}$$

So we get

$$\begin{aligned}
d_{TV}(X^t, Y^t) &\leq \Pr\left[X^t \neq Y^t\right] \\
&\leq \left(1 - \frac{1}{n} \cdot \frac{k-2\Delta}{k-\Delta}\right)^t \cdot \mathrm{E}\left[d(X^0, Y^0)\right] \\
&\leq \exp\left(-\frac{t}{n} \cdot \frac{k-2\Delta}{k-\Delta}\right) \cdot n.
\end{aligned}$$

For fixed $k > 2\Delta$, this is $e^{-\Theta(t/n)}n$, which will be less than ϵ for $t = \Omega(n(\log n + \log(1/\epsilon)))$.

Curious, when $k = 2\Delta$, we can still use this argument to prove convergence, since in this case the expected change in $d(X, Y)$ at each step is at most 0, and it changes by 1 or more with probability $\Omega(1/n)$. So we have a random walk that is at worst unbiased and at best biased in the direction we like. The convergence bound is not so good—$O(n^3 \log(1/\epsilon))$—but it is still polylogarithmic in the size of the state space.

For $k < 2\Delta$, the argument collapses: it's more likely than not that X and Y move away from each other, so even if X's distribution converges to the stationary distribution, we won't see X and Y coalesce any time soon.

9.4.6.2 Sampling independent sets

For a more complicated example of path coupling, let's try sampling independent sets of vertices on a graph $G = (V, E)$ with n vertices and m edges. If we can bias in favor of larger sets, we might even get a good independent set approximation! The fact that this is hard will console us when we find that it doesn't work.

A natural way to set up the random walk is to represent each potentially independent set as a bit vector, where 1 indicates membership in the set,

and at each step we pick one of the n bits uniformly at random and set it to 0 or 1 with probability 1/2 each, provided that the resulting set is independent. If the resulting set is not independent, we stay put.[11]

It's easy to see that $d(x, y) = \|x - y\|_1$ is a bound on the length of the minimum number of transitions to get from x to y, since we can alway remove all the extra ones from x and put back the extra ones in y while preserving independence throughout the process. (This argument also shows that the Markov chain is irreducible.) It may be hard to find the exact minimum path length, so we'll use this distance instead for our path coupling.

We can easily show that the stationary distribution of this process is uniform. The essential idea is that if we can transform one independent set S into another S' by flipping a bit, then we can go back by flipping the bit the other ways. Since each transition happens with the same probability $1/n$, we get $\pi_S \cdot (1/n) = \pi_{S'} \cdot (1/n)$ and $\pi_S = \pi_{S'}$. Since we can apply this equation along a path between any two states, all states must have the same probability in the unique stationary distribution.

To prove convergence, it's tempting to start with the obvious coupling, even though it doesn't actually work. Pick the same position and value for both copies of the chain. If x and y are adjacent, then they coalesce with probability $1/n$ (both probability $1/2n$ transitions are feasible for both copies, since the neighboring nodes always have the same state). What is the probability that they diverge? We can only be prevented from picking a value if the value is 1 and some neighbor is 1. So the bad case is when $x_i = 1$, $y_i = 0$, and we attempt to set some neighbor of i to 1; in the worst case, this happens $\Delta/2n$ of the time, which is at least $1/n$ when $\Delta \geq 2$. No coalescence here!

We now have two choices. We can try to come up with a more clever random walk (see [MU05, §11.6], or skip ahead a few paragraphs), or we can try to persevere with our dumb random walk, a more clever analysis, and possibly a more restricted version of the problem where it will miraculously work even though it shouldn't really. Let's start with $\Delta = 2$. Here the path coupling argument gives no expected change in $d(X_t, Y_t)$, so we have some hope that with enough wandering around they do in fact collide.

How do we prove this? Given arbitrary X_t, Y_t, we know from the path coupling argument above that on average they don't move any farther away. We also know that that there is a probability of at least $1/n$ that they move closer if they are not already identical (this is slightly tricky to see in the

[11] In statistical physics, this process of making a local change with probability proportional to how much we like the result goes by the name of **Glauber dynamics**.

general case; basically we always have a $1/2n$ chance of removing an extra 1 from one or the other, and if we also have an extra 1 in the other process, we get another $1/2n$, and if we don't, we can put in a missing 1 and get $1/2n$ that way instead). And we know that any change resulting from our not-very-smart coupling will change the distance by either 0 or ± 1. So if we sample only at times when a move has just occurred, we see a random walk (with a possible downward bias) with a reflecting barrier at n and an absorbing barrier at 0: this converges in n^2 steps. But since our random walk steps may take an expected n real steps each, we get a bound of n^3 on the total number of steps to converge.

But if $\Delta = 3$, we seem to be doomed. We are also in trouble if we try to bias the walk in favor of adding vertices: since our good case is a $1 \to 0$ transition, decreasing its probability breaks the very weak balance we have even with $\Delta = 2$ (for $\Delta = 1$, it's not a problem, but this is an even less interesting case) So maybe we should see what we can do with the sneakier random walk described in [MU05, §11.6] (which is originally due to Luby and Vigoda [LV99]).

Here the idea is that we pick a random edge uv, and then try to do one of the following operations, all with equal probability:

1. Set $u = v = 0$.
2. Set $u = 0$ and $v = 1$.
3. Set $u = 1$ and $v = 0$.

In each case, if the result would be a non-independent set, we instead do nothing.

Verifying that this has a uniform stationary distribution is mildly painful if we are not careful, since there may be several different transitions that move from some state x to the same state y. But for each transition (occurring with probability $\frac{1}{3m}$), we can see that there is a reverse transition that occurs with equal probability; so the detailed balance equations (9.3.1) hold with uniform probabilities. Note that we can argue this even though we don't know what the actual stationary probabilities are, since we don't know how many independent sets our graph has.

So now what happens if we run two coupled copies of this process, where the copies differ on exactly one vertex i?

First, every neighbor of i is 0 in both processes. A transition that doesn't involve any neighbors of i will have the same effect on both processes. So we need to consider all choices of edges where one of the endpoints is either

i or a neighbor j of i. In the case where the other endpoint isn't i, we'll call it k; there may be several such k.

If we choose ij and don't try to set j to one, we always coalesce the states. This occurs with probability $\frac{2}{3m}$. If we try to set i to zero and j to one, we may fail in both processes, because j may have a neighbor k that is already one; this will preserve the distance between the two processes. Similarly, if we try to set j to one as part of a change to some jk, we will also get a divergence between the two processes: in this case, the distance will actually increase. This can only happen if j has at most one neighbor k (other than i) that is already in the independent set; if there are two such k, then we can't set j to one no matter what the state of i is.

This argument suggests that we need to consider three cases for each j, depending on the number s of nodes $k \neq i$ that are adjacent to j and have $x_k = y_k = 1$. In each case, we assume $x_i = 0$ and $y_i = 1$, and that all other nodes have the same value in both x and y. (Note that these assumptions mean that any such k can't be adjacent to i, because we have $y_k = y_i = 1$.)

- $s = 0$. Then if we choose ij, we can always set i and j however we like, giving a net $-\frac{1}{m}$ expected change to the distance. However, this is compensated for by up to $d-1$ attempts to set $j = 1$ and $k = 0$ for some k, all of which fail in one copy of the process but succeed in the other. Since k doesn't change, each of these failures adds only 1 to the distance, which becomes at most $\frac{d-1}{3m}$ total. So our total expected change in this case is at most $\frac{d-4}{3m}$.

- $s = 1$. Here attempts to set $i = 0$ and $j = 1$ fail in both processes, giving only a $-\frac{2}{3m}$ expected change after picking ij. Any change to jk fails only if we set $j = 1$, which we can only do in the x process and only if we also set $k = 0$ for the unique k that is currently one. This produces an increase in the distance of 2 with probability $\frac{1}{3m}$, exactly canceling out the decrease from picking ij. Total expected change is 0.

- $s = 2$. Now we can never set $j = 1$. So we drop $-\frac{2}{3m}$ from changes to ij and have no change in distance from updates to jk for any $k \neq i$.

Considering all three cases, if $\Delta \leq 4$, then in the worst case we have $\mathrm{E}\left[d(X_{t+1}, Y_{t+1} \mid X_t, Y_t\right] = d(X_t, Y_t)$. We also have that the distance changes with probability at least $\frac{2}{3m}$. So the same analysis as for the dumb process shows that we converge in at most $\frac{3}{8}n^2 m$ steps on average.

Here, we've considered the case where all independent sets have the same probability. One can also bias the random walk in favor of larger independent sets by accepting increases with higher probability than decreases (as in Metropolis-Hastings); this samples independent sets of size s with probability proportional to λ^s. Some early examples of this approach are given in [LV97, LV99, DG00]. The question of exactly which values of λ give polynomial convergence times is still open; see [MWW07] for some recent bounds.

9.4.6.3 Metropolis-Hastings and simulated annealing

Recall that the Metropolis-Hastings algorithm constructs a reversible Markov chain with a desired stationary distribution from any reversible Markov chain on the same states (see §9.3.3 for details.)

A variant, which generally involves tinkering with the chain while it's running, is the global optimization heuristic known as **simulated annealing**. Here we have some function g that we are trying to minimize. So we set $f(i) = \exp(-\alpha g(i))$ for some $\alpha > 0$. Running Metropolis-Hastings gives a stationary distribution that is exponentially weighted to small values of g; if i is the global minimum and j is some state with high $g(j)$, then $\pi(i) = \pi(j)\exp(\alpha(g(j) - g(i))$, which for large enough α goes a long way towards compensating for the fact that in most problems there are likely to be exponentially more bad j's than good i's. The problem is that the same analysis applies if i is a local minimum and j is on the boundary of some depression around i; large α means that it is exponentially unlikely that we escape this depression and find the global minimum.

The simulated annealing hack is to vary α over time; initially, we set α small, so that we get conductance close to that of the original Markov chain. This gives us a sample that is roughly uniform, with a small bias towards states with smaller $g(i)$. After some time we increase α to force the process into better states. The hope is that by increasing α slowly, by the time we are stuck in some depression, it's a deep one—optimal or close to it. If it doesn't work, we can randomly restart and/or decrease α repeatedly to jog the chain out of whatever depression it is stuck in. How to do this effectively is deep voodoo that depends on the structure of the underlying chain and the shape of $g(i)$, so most of the time people who use simulated annealing just try it out with some generic **annealing schedule** and hope it gives some useful result. (This is what makes it a heuristic rather than an algorithm. Its continued survival is a sign that it does work at least sometimes.)

Here are toy examples of simulated annealing with provable convergence times.

Single peak Let's suppose x is a random walk on an n-dimensional hypercube (i.e., n-bit vectors where we set 1 bit at a time), $g(x) = |x|$, and we want to maximize g. Now a transition that increase $|x|$ is accepted always and a transition that decreases $|x|$ is accepted only with probability $e^{-\alpha}$. For large enough α, this puts a constant fraction of π on the single peak at $x = \mathbf{1}$; the observation is that that there are only $\binom{n}{k} \leq n^k$ points with k zeros, so the total weight of all points is at most $\pi(\mathbf{1}) \sum_{k \geq 0} n^k \exp(-\alpha k) = \pi(\mathbf{1}) \sum \exp(\ln n - \alpha)^k = \pi(\mathbf{1})/(1 - \exp(\ln n - \alpha)) = \pi(\mathbf{1}) \cdot O(1)$ when $\alpha > \ln n$, giving $\pi(\mathbf{1}) = \Omega(1)$ in this case.

So what happens with convergence? Let $p = \exp(-\alpha)$. Let's try doing a path coupling between two adjacent copies x and y of the Metropolis-Hastings process, where we first pick a bit to change, then pick a value to assign to it, accepting the change in both processes if we can. The expected change in $|x - y|$ is then $(1/2n)(-1 - p)$, since if we pick the bit where x and y differ, we have probability $1/2n$ of setting both to 1 and probability $p/2n$ of setting both to 0, and if we pick any other bit, we get the same distribution of outcomes in both processes. This gives a general bound of $\mathrm{E}\left[|X_{t+1} - Y_{t+1}| \mid |X_t - Y_t|\right] \leq (1 - (1+p)/2n)|X_t - Y_t|$, from which we have $\mathrm{E}\left[|X_t - Y_t|\right] \leq \exp(-t(1+p)/2n)\, \mathrm{E}\left[|X_0 - Y_0|\right] \leq n \exp(-t(1+p)/2n)$. So after $t = 2n/(1+p) \ln(n/\epsilon)$ steps, we expect to converge to within ϵ of the stationary distribution in total variation distance. This gives an $O(n \log n)$ algorithm for finding the peak.

This is kind of a silly example, but if we suppose that g is better disguised (for example, $g(x)$ could be $|x \oplus r|$ where r is a random bit vector), then we wouldn't really expect to do much better than $O(n)$. So $O(n \log n)$ is not bad for an algorithm with no brains at all.

Single peak with very small amounts of noise Now we'll let $g : 2^n \to \mathbb{N}$ be some arbitrary Lipschitz function (in this case we are using the real definition: $|g(x) - g(y)| \leq |x - y|$) and ask for what values of $p = e^{-\alpha}$ the Metropolis-Hastings walk with $f(i) = e^{-\alpha g(i)}$ can be shown to converge quickly. Given adjacent states x and y, with $x_i \neq y_i$ but $x_j = y_j$ for all $j \neq i$, we still have a probability of at least $(1+p)/2n$ of coalescing the states by setting $x_i = y_i$. But now there is a possibility that if we try to move to $(x[j/b], y[j/b])$ for some j and b, that x rejects while y does not or vice versa (note if $x_j = y_j = b$, we don't move in either copy of

the process). Conditioning on j and b, this occurs with probability $1-p$ precisely when $x[j/b] < x$ and $y[j/b] \geq y$ or vice versa, giving an expected increase in $|x-y|$ of $(1-p)/2n$. We still get an expected net change of $-2p/2n = -p/n$ provided there is only one choice of j and b for which this occurs. So we converge in time $\tau(\epsilon) \leq (n/p)\log(n/\epsilon)$ in this case.[12]

One way to think of this is that the shape of the neighborhoods of nearby points is similar. If I go up in a particular direction from point x, it's very likely that I go up in the same direction from some neighbor y of x.

If there are more bad choices for j and b, then we need a much larger value of p: the expected net change is now $(k(1-p)-1-p)/2n = (k-1-(k+1)p)/2n$, which is only negative if $p > (k-1)/(k+1)$. This gives much weaker pressure towards large values of g, which still tends to put us in high neighborhoods but creates the temptation to fiddle with α to try to push us even higher once we think we are close to a peak.

9.5 Spectral methods for reversible chains

(See also `http://www.stat.berkeley.edu/~aldous/RWG/Chap3.pdf`, from which many of the details in the notes below are taken.)

The problem with coupling is that (a) it requires cleverness to come up with a good coupling; and (b) in many cases, even that doesn't work—there are problems for which no coupling that only depends on current and past transitions coalesces in a reasonable amount of time.[13] When we run into these problems, we may be able to show convergence instead using a linear-algebraic approach, where we look at the eigenvalues of the transition matrix of our Markov chain. This approach works best for reversible Markov chains, where $\pi_i p_{ij} = \pi_j p_{ij}$ for all states i and j and some distribution π.

9.5.1 Spectral properties of a reversible chain

Suppose that P is the transition matrix of an irreducible, reversible Markov chain. Then it has a unique stationary distribution π that is a left **eigenvector** corresponding to the **eigenvalue** 1, which just means that $\pi P = 1\pi$. For aperiodic chains, P will have a total of n real eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$,

[12] You might reasonably ask if such functions g exist. One example is $g(x) = (x_1 \oplus x_2) + \sum_{i>2} x_i$.

[13] Such a coupling is called a **causal coupling**; an example of a Markov chain for which causal couplings are known not to work is the one used for sampling perfect matchings in bipartite graphs as described in §9.5.8.4 [KR99].

where $\lambda_1 = 1$ and $\lambda_n > -1$. Each eigenvalue λ_i has a corresponding eigenvector u^i, a nonzero vector that satisfies $u^i P = \lambda_i u^i$. In Markov chain terms, these eigenvectors correspond to deviations from the stationary distribution that shrink over time.

For example, the transition matrix

$$S = \begin{bmatrix} p & q \\ q & p \end{bmatrix}$$

corresponding to a Markov chain on two states that stays in the same state with probability p and switches to the other state with probability $q = 1 - p$ has eigenvectors $u_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$ and $u_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}$ with corresponding eigenvalues $\lambda_1 = 1$ and $\lambda_2 = p - q$, as shown by computing

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} p & q \\ q & p \end{bmatrix} = \begin{bmatrix} p+q & q+p \end{bmatrix} = 1 \cdot \begin{bmatrix} 1 & 1 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} p & q \\ q & p \end{bmatrix} = \begin{bmatrix} p-q & q-p \end{bmatrix} = (p-q) \cdot \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

9.5.2 Analysis of symmetric chains

To make our life easier, we will assume that in addition to being reversible, our Markov chain has a uniform stationary distribution. Then $\pi_i = \pi_j$ for all i and j, and so reversibility implies $p_{ij} = p_{ji}$ for all i and j as well, meaning that the transition matrix P is symmetric. Symmetric matrices have the very nice property that their eigenvectors are all orthogonal (this is the **spectral theorem**), which allows for a very straightforward decomposition of the distribution on the initial state (represented as a vector x_0) as a linear combination of the eigenvectors. For example, if initially we put all our weight on state 1, we get $x_0 = \begin{bmatrix} 1 & 0 \end{bmatrix} = \frac{1}{2}u^1 + \frac{1}{2}u^2$.

If we now take a step in the chain, we multiply x by P. We can express this as

$$\begin{aligned} xP &= \left(\frac{1}{2}u^1 + \frac{1}{2}u^2\right)P \\ &= \frac{1}{2}u^1 P + \frac{1}{2}u^2 P \\ &= \frac{1}{2}\lambda_1 u^1 + \frac{1}{2}\lambda_2 u^2. \end{aligned}$$

This uses the defining property of eigenvectors, that $u^i P = \lambda_i u^i$.

In general, if $x = \sum_i a_i u^i$, then $xP = \sum_i a_i \lambda_i u^i$ and $xP^t = \sum a_i \lambda_i^t u^i$. For any eigenvalue λ_i with $|\lambda_i| < 1$, λ_i^t goes to zero in the limit. So only those eigenvectors with $\lambda_i = 1$ survive. For aperiodic chains, these consist only of the stationary distribution $\pi = u^i / \|u^i\|_1$. For periodic chains, there may be some additional eigenvalues with $|\lambda_i| = 1$, but the only possibility that arises for a reversible chain is $\lambda_n = -1$, corresponding to a chain with period 2.[14]

Assuming that $|\lambda_2| \geq |\lambda_n|$, as t grows large λ_2^t will dominate the other smaller eigenvalues, and so the size of λ_2 will control the rate of convergence of the underlying Markov process.

This assumption is always true for lazy walks that stay put with probability 1/2, because we can think of such a walk as having transition matrix $\frac{1}{2}(P + I)$, where I is the identity matrix and P is the transition matrix of the unlazy version of the walk. If $xP = \lambda x$ for some x and λ, then $x\Big(\frac{1}{2}(P + I)\Big) = \Big(\frac{1}{2}(\lambda + 1)\Big)x$. This means that x is still an eigenvector of the lazy walk, and its corresponding eigenvalue is $\frac{1}{2}(\lambda+1) \geq \frac{1}{2}((-1)+1) \geq 0$.

But what does having small λ_2 mean for total variation distance? If x^t is a vector representing the distribution of our position at time t, then $d_{TV} = \frac{1}{2}\sum_{i=1}^n |x_i^t - \pi_i| = \frac{1}{2}\|x^t - \pi\|_1$. But we know that $x^0 = \pi + \sum_{i=2}^n c_i u^i$ for some coefficients c_i, and $x^t = \pi + \sum_{i=2}^n \lambda_i^t c_i u^i$. So we are looking for a bound on $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_1$.

It turns out that it is easier to get a bound on the ℓ_2 norm $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_2$. Here we use the fact that the eigenvectors are orthogonal. This means that the Pythagorean theorem holds and $\|\sum_{i=2}^n \lambda_i^t c_i u^i\|_2^2 = \sum_{i=2}^n \lambda_i^{2t} c_i^2 \|u^i\|_2^2 =$

[14] Chains with periods greater than 2 (which are never reversible) have pairs of complex-valued eigenvalues that are roots of unity, which happen to cancel out to only produce real probabilities in vP^t. Chains that aren't irreducible will have one eigenvector with eigenvalue 1 for each final component; the stationary distributions of these chains are linear combinations of these eigenvectors (which are just the stationary distributions on each component).

$\sum_{i=2}^{n} \lambda_i^{2t} c_i^2$ if we normalize each u^i so that $\|u^i\|_2^2 = 1$. But then

$$\begin{aligned}
\left\|x^t - \pi\right\|_2^2 &= \sum_{i=2}^{n} \lambda_i^{2t} c_i^2 \\
&\leq \sum_{i=2}^{n} \lambda_2^{2t} c_i^2 \\
&= \lambda_2^{2t} \sum_{i=2}^{n} c_i^2 \\
&= \lambda_2^{2t} \|x^0 - \pi\|_2^2.
\end{aligned}$$

Now take the square root of both sides to get

$$\left\|x^t - \pi\right\|_2 =< \lambda_2^t \left\|x^0 - \pi\right\|.$$

To translate this back into d_{TV}, we use the inequality

$$\|x\|_2 \leq \|x\|_1 \leq \sqrt{n}\|x\|_2,$$

which holds for any n-dimensional vector x. Because $\|x^0\|_1 = \|\pi\|_1 = 1$, $\|x^0 - \pi\|_1 \leq 2$ by the triangle inequality, which also gives $\|x^0 - \pi\|_2 \leq 2$. So

$$\begin{aligned}
d_{TV}(x^t, \pi) &= \frac{1}{2}\|x^t - \pi\|_1 \\
&\leq \frac{\sqrt{n}}{2} \lambda_2^t \|x^0 - \pi\|_2 \\
&\leq \lambda_2^t \sqrt{n}.
\end{aligned}$$

If we want to get $d_T V(x^t, n) \leq \epsilon$, we will need $t \ln(\lambda_2) + \ln \sqrt{n} \leq \ln \epsilon$ or $t \geq \frac{1}{\ln(1/\lambda_2)} \left(\frac{1}{2} \ln n + \ln(1/\epsilon)\right)$.

The factor $\frac{1}{\ln(1/\lambda_2)} = \frac{1}{-\ln \lambda_2}$ can be approximated by $\tau_2 = \frac{1}{1-\lambda_2}$, which is called the **mixing rate** or **relaxation time** of the Markov chain. Indeed, our old friend $1 + x \leq e^x$ implies $\ln(1 + x) \leq x$, which gives $\ln \lambda_2 \leq \lambda_2 - 1$ and thus $\frac{1}{-\ln \lambda_2} \leq \frac{1}{1-\lambda_2} = \tau_2$. So $\tau_2 \left(\frac{1}{2} \ln n + \ln(1/\epsilon)\right)$ gives a conservative estimate on the time needed to achieve $d_T V(x^t, \pi) \leq \epsilon$ starting from an arbitrary distribution.

9.5.3 Analysis of asymmetric chains

If the stationary distribution is not uniform, then in general the transition matrix will not be symmetric. We can make it symmetric by scaling the

probability of being in the i-th state by $\pi_i^{-1/2}$. The idea is to decompose our transition matrix P as $\Pi A \Pi^{-1}$, where Π is a diagonal matrix with $\Pi_{ii} = \pi_i^{-1/2}$, and $A_{ij} = \sqrt{\frac{\pi_j}{\pi_i}} P_{ij}$. Then A is symmetric, because

$$\begin{aligned} A_{ij} &= \sqrt{\frac{\pi_j}{\pi_i}} P_{ij} \\ &= \sqrt{\frac{\pi_j}{\pi_i}} \frac{\pi_i}{\pi_j} P_{ji} \\ &= \sqrt{\frac{\pi_i}{\pi_j}} P_{ji} \\ &= A_{ji}. \end{aligned}$$

So now given an initial distribution x_0, we can apply the same reasoning as before to show that $\|\pi\Pi - x\Pi\|_2$ shrinks by λ_2 with every step of the Markov chain. The difference now is that the initial distance is affected by the scaling we did in Π; so instead of getting $d_{TV}(x^t, \pi) \leq \lambda_2^t \sqrt{n}$, we get $d_{TV}(x^t, \pi) \leq \lambda_2^t (\pi_{\min})^{-1/2}$, where $\pi_{\min}$ is the smallest probability of any single node in the stationary distribution π. The bad initial x_0 in this case is the one that puts all its weight on this node, since this maximizes its distance from π after scaling.

For more details on this, see [AF01, §3.4].

So now we just need a tool for bounding λ_2.

9.5.4 Conductance

The **conductance** or **Cheeger constant** to $\Phi(S)$ of a set S of states in a Markov chain is

$$\Phi(S) = \frac{\sum_{i \in S, j \notin S} \pi_i p_{ij}}{\pi(S)}. \tag{9.5.1}$$

This is the probability of leaving S on the next step starting from the stationary distribution conditioned on being in S. The conductance is a measure of how easy it is to escape from a set; it can also be thought of as a weighted version of edge expansion.

The conductance of a Markov chain as a whole is obtained by taking the minimum of $\Phi(S)$ over all S that occur with probability at most 1/2:

$$\Phi = \min_{0 < \pi(S) \leq 1/2} \Phi(S). \tag{9.5.2}$$

The usefulness of conductance is that it bounds λ_2:

Theorem 9.5.1. *In a reversible Markov chain,*

$$1 - 2\phi \leq \lambda_2 \leq 1 - \phi^2/2. \tag{9.5.3}$$

The bound (9.5.3) is known as the **Cheeger inequality**. We won't attempt to prove it here.

For lazy walks we always have $\lambda_2 = \lambda_{\max}$, and so we can convert this to a bound on the relaxation time:

Corollary 9.5.2.

$$\frac{1}{2\Phi} \leq \tau_2 \leq \frac{2}{\phi^2}. \tag{9.5.4}$$

In other words, high conductance implies low relaxation time and vice versa, up to squaring.

9.5.5 Easy cases for conductance

For very simple Markov chains we can compute the conductance directly. Consider a lazy random walk on a cycle. Any proper subset S has at least two outgoing edges, each of which carries a flow of $1/4n$, giving $\Phi_S \geq (1/2n)/\pi(S)$. If we now take the minimum of Φ_S over all S with $\pi(S) \leq 1/2$, we get $\phi \geq 1/n$, which gives $\tau_2 \leq 2n^2$. This is essentially the same bound as we got from coupling.

Here's a slightly more complicated chain. Take two copies of K_n, where n is odd, and join them by a path with n edges. Now consider Φ_S for a lazy random walk on this graph where S consists of half the graph, split at the edge in the middle of the path. There is a single outgoing edge uv, with $\pi(u) = d(u)/2|E| = 2/(2n(n-1)n/2 + n) = 2n^{-2}$ and $p_{uv} = 1/4$, for $\pi(u)p_{uv} = n^{-2}/2$. By symmetry, we have $\pi(S) \to 1/2$ as $n \to \infty$, giving $\Phi_S \to n^{-2}(1/2)/(1/2) = n^{-2}$. So we have $n^2/2 \leq \tau_2 \leq 2n^4$.

How does this compare to the actual mixing time? In the stationary distribution, we have a constant probability of being in each of the copies of K_n. Suppose we start in the left copy. At each step there is a $1/n$ chance that we are sitting on the path endpoint. We then step onto the path with probability $1/n$, and reach the other end before coming back with probability $1/n$. So (assuming we can make this extremely sloppy handwaving argument rigorous) it takes at least n^3 steps on average before we reach the other copy of K_n, which gives us a rough estimate of the mixing time of $\Theta(n^3)$. In this case the exponent is exactly in the middle of the bounds derived from conductance.

9.5.6 Edge expansion using canonical paths

Here and below we are mostly following the presentation in [Gur00], but with slightly different examples (and probably more errors).

For more complicated Markov chains, it is helpful to have a tool for bounding conductance that doesn't depend on intuiting what sets have the smallest boundary. The **canonical paths** [JS89] method does this by assigning a unique path γ_{xy} from each state x to each state y in a way that doesn't send too many paths across any one edge. So if we have a partition of the state space into sets S and T, then there are $|S| \cdot |T|$ paths from states in S to states in T, and since (a) every one of these paths crosses an S–T edge, and (b) each S–T edge carries at most ρ paths, there must be at least $|S| \cdot |T|/\rho$ edges from S to T. Note that because there is no guarantee we chose good canonical paths, this is only useful for getting lower bounds on conductance—and thus upper bounds on mixing time—but this is usually what we want.

Let's start with a small example. Let $G = C_n \,\square\, C_m$, the $n \times m$ torus. A lazy random walk on this graph moves north, east, south, or west with probability 1/8 each, wrapping around when the coordinates reach n or m. Since this is a random walk on a regular graph, the stationary distribution is uniform. What is the relaxation time?

Intuitively, we expect it to be $O(\max(n, m)^2)$, because we can think of this two-dimensional random walk as consisting of two one-dimensional random walks, one in the horizontal direction and one in the vertical direction, and we know that a random walk on a cycle mixes in $O(n^2)$ time. Unfortunately, the two random walks are not independent: every time I take a horizontal step is a time I am not taking a vertical step. We *can* show that the expected coupling time is $O(n^2 + m^2)$ by running two sequential instances of the coupling argument for the cycle, where we first link the two copies in the horizontal direction and then in the vertical direction. So this gives us one bound on the mixing time. But what happens if we try to use conductance?

Here it is probably easiest to start with just a cycle. Given points x and y on C_n, let the canonical path γ_{xy} be a shortest path between them, breaking ties so that half the paths go one way and half the other. Then each each is crossed by exactly k paths of length k for each $k = 1 \ldots (n/2 - 1)$, and at most $n/4$ paths of length $n/2$ (0 if n is odd), giving a total of $\rho \leq (n/2 - 1)(n/2)/2 + n/4 = n^2/8$ paths across the edge.

If we now take an S–T partition where $|S| = m$, we get at least $m(n - m)/\rho = 8m(n - m)/n^2$ S–T edges. This peaks at $m = n/2$, where we get

2 edges—exactly the right number—and in general when $m \leq n/2$ we get at least $8m(n/2)/n^2 = 4m/n$ outgoing edges, giving a conductance $\Phi_S \geq (1/4n)(4m/n)/(m/n) = 1/n$. (This is essentially what we got before, except we have to divide by 2 because we are doing a lazy walk. Note that for small m, the bound is a gross underestimate, since we know that every nonempty proper subset has at least 2 outgoing edges.)

Now let's go back to the torus $C_n \,\square\, C_m$. Given x and y, define γ_{xy} to be the L-shaped path that first changes x_1 to y_1 by moving the shortest distance vertically, then changes x_2 to y_2 by moving the shortest distance horizontally. For a vertical edge, the number of such paths that cross it is bounded by $n^2m/8$, since we get at most $n^2/8$ possible vertical path segments and for each such vertical path segment there are m possible horizontal destinations to attach to it. For a horizontal edge, n and m are reversed, giving $m^2n/8$ paths. To make our life easier, let's assume $n \geq m$, giving a maximum of $\rho = n^2m/8$ paths.

For $|S| \leq nm/2$, this gives at least

$$\frac{|S| \cdot |G \setminus S|}{\rho} \geq \frac{|S|(nm/2)}{n^2m/8} = \frac{4|S|}{n}$$

outgoing edges. We thus have

$$\Phi(S) \geq \frac{1}{8nm} \cdot \frac{4|S|/n}{|S|/nm} = \frac{1}{2n}.$$

This gives $\tau_2 \leq 2/\Phi^2 \leq 8n^2$. Given that we assumed $n \geq m$, this is essentially the same $O(n^2 + m^2)$ bound that we would get from coupling.

9.5.7 Congestion

For less symmetric chains, we weight paths by the probabilities of their endpoints when counting how many cross each edge, and treat the flow across the edge as a capacity. This gives the **congestion** of a collection of canonical paths $\Gamma = \{\gamma_{xy}\}$, which is computed as

$$\rho(\Gamma) = \max_{uv \in E} \frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y,$$

and we define $\rho = \min_\Gamma \rho(\Gamma)$.

The intuition here is that the congestion bounds the ratio between the canonical path flow across an edge and the Markov chain flow across the edge. If the congestion is not too big, this means that any cut that has a lot

of canonical path flow across it also has a lot of Markov chain flow. When $\pi(T) \geq 1/2$, the total canonical path flow $\pi(S)\pi(T)$ is at least $(1/2)\pi(S)$; so this means that when $\pi(S)$ is large but less than $1/2$, the Markov chain flow leaving S is also large. This gives a lower bound on conductance.

Formally, we have the following lemma:

Lemma 9.5.3.

$$\Phi \geq \frac{1}{2\rho} \tag{9.5.5}$$

from which it follows that

$$\tau_2 \leq 8\rho^2. \tag{9.5.6}$$

Proof. Pick some set of canonical paths Γ with $\rho(\Gamma) = \rho$. Now pick some S–T partition with $\phi(S) = \phi$. Consider a flow where we route $\pi(x)\pi(y)$ units of flow along each path γ_{xy} with $x \in S$ and $y \in T$. This gives a total flow of $\pi(S)\pi(T) \geq \pi(S)/2$. We are going to show that we need a lot of capacity across the S–T cut to carry this flow, which will give the lower bound on conductance.

For any S–T edge uv, we have

$$\frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \leq \rho$$

or

$$\sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \leq \rho \pi_u p_{uv}.$$

Since each S–T path crosses at least one S–T edge, we have

$$\begin{aligned}
\pi(S)\pi(T) &= \sum_{x \in S, y \in T} \pi_x \pi_y \\
&\leq \sum_{u \in S, t \in V, uv \in E} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \\
&\leq \sum_{u \in S, t \in V, uv \in E} \rho \pi_u p_{uv}. \\
&= \rho \sum_{u \in S, t \in V, uv \in E} \pi_u p_{uv}.
\end{aligned}$$

But then

$$\begin{aligned}\Phi(S) &= \frac{\sum_{u\in S, t\in V, uv\in E} \pi_u p_{uv}}{\pi_S} \\ &\geq \frac{\pi(S)\pi(T)/\rho}{\pi(S)} \\ &= \frac{\pi(T)}{\rho} \\ &\geq \frac{1}{2\rho}.\end{aligned}$$

To get the bound on τ_2, use (9.5.4) to compute $\tau_2 \leq 2/\phi^2 \leq 8\rho^2$. □

9.5.8 Examples

Here are some more examples of applying the spectral method.

9.5.8.1 Lazy random walk on a line

Consider a lazy random walk on a line with reflecting barriers at 0 and $n-1$. Here $\pi_x = \frac{1}{n}$ for all x. There aren't a whole lot of choices for canonical paths; the obvious choice for γ_{xy} with $x < y$ is $x, x+1, x+2, \ldots, y$. This puts $(n/2)^2$ paths across the middle edge (which is the most heavily loaded), each of which has weight $\pi_x\pi_y = n^{-2}$. So the congestion is $\frac{1}{(1/n)(1/4)}(n/2)^2 n^-2 = n$, giving a mixing time of at most $8n^2$. This is a pretty good estimate.

9.5.8.2 Random walk on a hypercube

Let's try a more complicated example: the random walk on a hypercube from §9.4.4. Here at each step we pick some coordinate uniformly at random and set it to 0 or 1 with equal probability; this gives a transition probability $p_{uv} = \frac{1}{2n}$ whenever i and j differ by exactly one bit. A plausible choice for canonical paths is to let γ_{xy} use **bit-fixing routing**, where we change bits in x to the corresponding bits in y from left to right. To compute congestion, pick some edge uv, and let k be the bit position in which u and v differ. A path γ_{xy} will cross uv if $u_k \ldots u_n = x_k \ldots x_n$ (because when we are at u we haven't fixed those bits yet) and $v_1 \ldots v_k = y_1 \ldots y_k$ (because at v we have fixed all of those bits). There are 2^{k-1} choices of $x_1 \ldots x_{k-1}$ consistent with the first condition and 2^{n-k} choices of $y_{k+1} \ldots y_n$ consistent with the second, giving exactly 2^{n-1} total paths γ_{xy} crossing uv. Since each path

occurs with weight $\pi_x \pi_y = 2^{-2n}$, and the flow across uv is $\pi_u p_{uv} = 2^{-n} \frac{1}{2n}$, we can calculate the congestion

$$\begin{aligned}\rho(\Gamma) &= \max_{uv \in E} \frac{1}{\pi_u p_{uv}} \sum_{\gamma_{xy} \ni uv} \pi_x \pi_y \\ &= \frac{1}{2^{-n}/2n} \cdot 2^{n-1} \cdot 2^{-2n} \\ &= n.\end{aligned}$$

This gives a relaxation time $\tau_2 \leq 8\rho^2 = 8n^2$. In this case the bound is substantially worse than what we previously proved using coupling.

The fact that the number of canonical paths that cross a particular edge is exactly one half the number of nodes in the hypercube is not an accident: if we look at what information we need to fill in to compute x and y from u and v, we need (a) the part of x we've already gotten rid of, plus (b) the part of y we haven't filled in yet. If we stitch these two pieces together, we get all but one of the n bits we need to specify a node in the hypercube, the missing bit being the bit we flip to get from u to v. This sort of thing shows up a lot in conductance arguments where we build our canonical paths by fixing a structure one piece at a time.

9.5.8.3 Matchings in a graph

A **matching** in a graph $G = (V, E)$ is a subset of the edges with no two elements adjacent to each other; equivalently, it's a subgraph that includes all the vertices in which each vertex has degree at most 1. We can use a random walk to sample matchings from an arbitrary graph uniformly.

Here is the random walk. Let S_t be the matching at time t. At each step, we choose an edge $e \in E$ uniformly at random, and flip a coin to decide whether to include it or not. If the coin comes up tails, set $S_{t+1} = S_t \setminus \{e\}$ (this may leave $S_{t+1} = S_t$ if S_t already omitted e); otherwise, set $S_{t+1} = S_t \cup \{e\}$ unless one of the endpoints of e is already incident to an edge in S_t, in which case set $S_{t+1} = S_t$.

Because this chain contains many self-loops, it's aperiodic. It's also straightforward to show that any transition between two adjacent matchings occurs with probability exactly $\frac{1}{2|E|}$, and thus that the chain is reversible with a uniform stationary distribution. We'd like to bound the congestion of the chain to show that it converges in a reasonable amount of time.

Let N be the number of matchings in G, and let $n = |V|$ and $m = |E|$ as usual. Then $\pi_S = 1/N$ for all S and $\pi_S p_{ST} = \frac{1}{2Nm}$. Our congestion for any

```
*--*  *    *  *  *    *  *  *    *  *--*    *  *--*    *  *--*    *  *--*
      | =>       | =>         =>         =>         =>         => |
*--*  *    *--*  *    *--*  *    *--*  *    *  *  *    *  *--*    *  *--*


     *  *--*    *--*--*    *--*  *    *--*  *    *--*  *    *--*  *
     |                     |     |    |     |    |     |          |
     *  *--*    *  *--*    *  *--*    *  *--*    *--*  *    *--*  *
```

Figure 9.1: Transforming one matching on a cycle to another

transition ST will then be $2NmN^{-2} = 2m/N$ times the number of paths that cross ST; ideally this number of paths will be at most N times some small polynomial in n and/or m.

Suppose we are trying to get from some matching X to another matching Y. The graph $X \cup Y$ has maximum degree 2, so each of its connected components is either a path or a cycle; in addition, we know that the edges in each of these paths or cycles alternate whether they come from X or Y, which among other things implies that the cycles are all even cycles.

We can transform X to Y by processing these components in a methodical way: first order the components (say, by the increasing identity of the smallest vertex in each), then for each component replace the X edges with Y edges. To do the replacement, we walk across the path or cycle adding Y edges in some fixed order after first deleting the adjacent X edges. If we are clever about choosing which Y edge to start with (basically always pick an edge with only one adjacent X edge if possible, and then use edge ordering to break ties) we can ensure that for each transition ST it holds that $(X \cup Y) \setminus (S \cup T)$ consists of a matching plus an extra edge.

Figure 9.1 shows an ASCII art version of this process applied to a cycle of length 6, together with a picture of $(X \cup Y) \setminus (S \cup T)$ at each S–T transition.

At each transition, we can turn $(X \cup Y) \setminus (S \cup T)$ into a matching by deleting one edge. So we can compute $X \cup Y$ knowing $S \cup T$ by supplying a matching ($\leq N$ choices) plus an extra edge ($\leq m$ choices). Once we know what $X \cup Y$ is, we can reconstruct X and Y by looking at which component we are on, assigning the edges in the earlier components to Y (because we must have done them already), assigning the edges in the later components to X (because we haven't done them yet), splitting the edges in our current component appropriately (because we know how far we've gotten), and then taking complements as needed with respect to $X \cup Y$ to get the remaining

edges for each of X and Y. Since there are at most Nm ways to do this, we get that the ST transition is covered by at most Nm canonical paths γ_{XY}.

Plugging this back into our previous formula, we get $\rho \leq (2m/N)(Nm) = m^2$, which gives $\tau_2 \leq 8m^4$.

9.5.8.4 Perfect matchings in dense bipartite graphs

(Basically doing [MR95, §11.3].)

A similar chain can be used to sample perfect matchings in dense bipartite graphs, as shown by Jerrum and Sinclair [JS89] based on an algorithm by Broder [Bro86] that turned out to have a bug in the analysis [Bro88].

A **perfect matching** of a graph G is a subgraph that includes all the vertices and gives each of them exactly one incident edge. We'll be looking for perfect matchings in a **bipartite graph** consisting of n left vertices $u_1, \ldots, u_n$ and n right vertices $v_1, \ldots, v_n$, where every edge goes between a left vertex and a right vertex. We'll also assume that the graph is **dense**, which in this case means that every vertex has at least $n/2$ neighbors. This density assumption was used in the original Broder and Jerrum-Sinclair papers but removed in a later paper by Jerrum, Sinclair, and Vigoda [JSV04].

The random walk is similar to the random walk from the previous section restricted to the set of matchings with either $n-1$ or n edges. At each step, we first flip a coin (the usual lazy walk trick); if it comes up heads, we choose an edge uv uniformly at random and apply one of the following transformations depending on how uv fits in the current matching m_t:

1. If $uv \in m_t$, and $|m_t| = n$, set $m_{t+1} = m_t \setminus \{uv\}$.
2. If u and v are both unmatched in m_t, and $|m_t| = n-1$, set $m_{t+1} = m_t \cup \{uv\}$.
3. If exactly one of u and v is matched to some other node w, and $|m_t| = n-1$, perform a rotation that deletes the w edge and adds uv.
4. If none of these conditions hold, set $m_{t+1} = m_t$.

The walk can be started from any perfect matching, which can be found in $O(n^{5/2})$ time using a classic algorithm of Hopcroft and Karp [HK73] that repeatedly searches for an **augmenting path**, which is a path in G between two unmatched vertices that alternates between edges not in the matching and edges in the matching. (We'll see augmenting paths again below when we show that any near-perfect matching can be turned into a perfect matching using at most two transitions.)

We can show that this walk converges in polynomial time using a canonical path argument. This is done in two stages: first, we define canonical paths between all perfect matchings. Next, we define a short path from any matching of size $n-1$ to some nearby perfect matching, and build paths between arbitrary matchings by pasting one of these short paths on one or both ends of a long path between perfect matchings. This gives a collection of canonical paths that we can show to have low congestion.

To go between perfect matchings X and Y, consider $X \cup Y$ as a collection of paths and even cycles as in §9.5.8.3. In some standard order, fix each path cycle by first deleting one edge to make room, then using rotate operations to move the rest of the edges, then putting the last edge back in. For any transition S–T along the way, we can use the same argument that we can compute $X \cap Y$ from $S \cap T$ by supplying the missing edges, which will consist of a matching of size n plus at most one extra edge. So if N is the number of matchings of size n or $n-1$, the same argument used previously shows that at most Nm of these long paths cross any S–T transition.

For the short paths, we must use the density property. The idea is that for any matching that is not perfect, we can find an augmenting path of length at most 3 between two unmatched nodes on either side. Pick some unmatched nodes u and v. Each of these nodes is adjacent to at least $n/2$ neighbors; if any of these neighbors are unmatched, we just found an augmenting path of length 1. Otherwise the $n/2$ neighbors of u and the $n/2$ nodes matched to neighbors of v overlap (because v is unmatched, leaving at most $n-1$ matched nodes and thus at most $n/2-1$ nodes that are matched to something that's not a neighbor of v). So for each matching of size $n-1$, in at most two steps (rotate and then add an edge) we can reach some specific perfect matching. There are at most m^2 ways that we can undo this, so each perfect matching is associated with at most m^2 smaller matchings. This blows up the number of canonical paths crossing any transition by roughly m^4; by counting carefully we can thus show congestion that is $O(m^6)$ ($O(m^4)$ from the blow-up, m from the m in Nm, and m from $1/p_{ST}$).

It follows that for this process, $\tau_2 = O(m^{12})$. (I think a better analysis is possible.)

As noted earlier, this is an example of a process for which causal coupling doesn't work in less than exponential time [KR99], a common problem with Markov chains that don't have much symmetry. So it's not surprising that stronger techniques were developed specifically to attack this problem.

Chapter 10

Approximate counting

(See [MR95, Chapter 11] for details.)

Basic idea: we have some class of objects, we want to know how many of them there are. Ideally we can build an algorithm that just prints out the exact number, but for many problems this is hard.

A **fully polynomial-time randomized approximation scheme** or **FPRAS** for a numerical problem outputs a number that is between $(1-\epsilon)$ and $(1+\epsilon)$ times the correct answer, with probability at least 3/4 (or some constant bounded away from 1/2—we can amplify to improve it), in time polynomial in the input size n and $(1/\epsilon)$. In this chapter, we'll be hunting for FPRASs. But first we will discuss briefly why we can't just count directly in many cases.

10.1 Exact counting

A typical application is to a problem in the complexity class #**P**, problems that involve counting the number of accepting computation paths in a nondeterministic polynomial-time Turing machine. Equivalently, these are problems that involve counting for some input x the number of values r such that $M(x,r)=1$, where M is some machine in **P**. An example would be the problem **#SAT** of counting the number of satisfying assignments of some CNF formula.

The class #**P** (which is usually pronounced **sharp P** or **number P**) was defined by Leslie Valiant in a classic paper [Val79]. The central result in this paper is **Valiant's theorem**. This shows that any problem in #**P** can be **reduced** (by **Cook reductions**, meaning that we are allowed to use the target problem as a subroutine instead of just calling it once) to the

problem of computing the **permanent** of a square 0–1 matrix A, where the permanent is given by the formula $\sum_\pi \prod_i A_{i,\pi(i)}$, where the sum ranges over all $n!$ permutations π of the indices of the matrix. An equivalent problem is counting the number of **perfect matchings** (subgraphs including all vertices in which every vertex has degree exactly 1) of a bipartite graph. Other examples of #**P**-complete problems are #SAT (defined above) and **#DNF** (like #SAT, but the input is in DNF form; #SAT reduces to #DNF by negating the formula and then subtracting the result from 2^n).

Exact counting of #**P**-hard problems is likely to be very difficult: **Toda's theorem** [Tod91] says that being able to make even a single query to a #**P**-oracle is enough to solve any problem in the **polynomial-time hierarchy**, which contains most of the complexity classes you have probably heard of. Nonetheless, it is often possible to obtain good approximations to such problems.

10.2 Counting by sampling

If many of the things we are looking for are in the target set, we can count by sampling; this is what poll-takers do for a living. Let U be the universe we are sampling from and G be the "good" set of points we want to count. Let $\rho = |G|/|U|$. If we can sample uniformly from U, then we can estimate ρ by taking N independent samples and dividing the number of samples in G by N. This will give us an answer $\hat{\rho}$ whose expectation is ρ, but the accuracy may be off. Since the variance of each sample is $\rho(1-\rho) \approx \rho$ (when ρ is small), we get $\operatorname{Var}\left[\sum X_i\right] \approx m\rho$, giving a standard deviation of $\sqrt{m\rho}$. For this to be less than our allowable error $\epsilon m\rho$, we need $1 \leq \epsilon\sqrt{m\rho}$ or $N \geq \frac{1}{\epsilon^2\rho}$. This gets bad if ρ is exponentially small. So if we are to use sampling, we need to make sure that we only use it when ρ is large.

On the positive side, if ρ is large enough we can easily compute how many samples we need using Chernoff bounds. The following lemma gives a convenient estimate; it is based on [[MR95, Theorem 11.1]] with a slight improvement on the constant:

Lemma 10.2.1. *Sampling N times gives relative error ϵ with probability at least $1-\delta$ provided $\epsilon \leq 1.81$ and*

$$N \geq \frac{3}{\epsilon^2\rho} \ln \frac{2}{\delta}. \tag{10.2.1}$$

Proof. Suppose we take N samples, and let X be the total count for these

samples. Then $\mathrm{E}[X] = \rho N$, and (5.2.7) gives (for $\epsilon \leq 1.81$):

$$\Pr[|X - \rho N| \geq \epsilon \rho N] \leq 2e^{-\rho N \epsilon^2/3}.$$

Now set $2e^{-\rho N \epsilon^2/3} \leq \delta$ and solve for N to get (10.2.1). □

10.3 Approximating #DNF

(Basically just repeating presentation in [MR95, §11.2] of covering technique from Karp and Luby [KL85].)

A **DNF formula** is a formula that is in **disjunctive normal form**: it is an OR of zero or more **clauses**, each of which is an AND of variables or their negations. An example would be $(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge x_4) \vee x_2)$. The **#DNF** problem is to count the number of satisfying assignments of a formula presented in disjunctive normal form.

Solving #DNF exactly is #**P**-complete, so we don't expect to be able to do it. Instead, we'll get a FPRAS by cleverly sampling solutions. The need for cleverness arises because just sampling solutions directly by generating one of the 2^n possible assignments to the n variables may find no satisfying assignments at all.

The trick is to sample pairs (x, i), where x is an assignment that satisfies clause C_i. Let S be the set of such pairs. For each pair (x, i), define $f(x, i) = 1$ if and only if $C_j(x) = 0$ for all $j < i$. Then $\sum_{(x,i)\in S} f(x, i)$ counts every satisfying assignment x, because (a) there exists some i such that x satisfies C_i, and (b) only the smallest such i will have $f(x, i) = 1$. In effect, f is picking out a single canonical satisfied clause from each satisfying assignment; note also that we can compute f quickly by testing x against all clauses C_j with $j < i$.

Our goal is to estimate the proportion ρ of "good" pairs with $f(x, i) = 1$ out of all pairs in S, and then use this to estimate $\sum_{(x,i)\in S} f(x, i) = \rho|S|$. If we can sample from S uniformly, the proportion ρ of "good" pairs with $f(x, i) = 1$ is at least $1/m$, because every satisfying assignment x contributes at most m pairs total, and one of them is good.

The only tricky part is figuring out how to sample pairs (x, i) with $C_i(x) = 1$ so that all pairs occur with the same probability. Let $U_i = \{(x, i) \mid C_i(x) = 1\}$. Then we can compute $|U_i| = 2^{n-k_i}$ where k_i is the number of literals in C_i. Using this information, we can sample i first with probability $|U_i|/\sum_j |U_j|$, then sample x from U_i just by picking values for the $n - k$ variables not fixed by C_i.

With $N \geq \frac{4}{\epsilon^2(1/m)} \ln \frac{2}{\delta} = \frac{4m}{\epsilon^2} \ln \frac{2}{\delta}$, we obtain an estimate $\hat{\rho}$ for the proportion of pairs (x, i) with $f(x, i) = 1$ that is within ϵ relative error of ρ with probability at least $1 - \delta$. Multiplying this by $\sum |U_i|$ then gives the desired count of satisfying assignments.

It's worth noting that there's nothing special about DNF formulas in this method. Essentially the same trick will work for estimating the size of the union of any collection of sets U_i where we can (a) compute the size of each U_i; (b) sample from each U_i individually; and (c) test membership of our sample x in U_j for $j < i$.

10.4 Approximating #KNAPSACK

Here is an algorithm for approximating the number of solutions to a **KNAPSACK** problem, due to Dyer [Dye03]. We'll concentrate on the simplest version, 0–1 KNAPSACK, following the analysis in Section 2.1 of [Dye03].

For the 0–1 KNAPSACK problem, we are given a set of n objects of weight $0 \leq a_1 \leq a_2 \leq \ldots a_n \leq b$, and we want to find a 0–1 assignment $x_1, x_2, \ldots, x_n$ such that $\sum_{i=1}^{n} a_i x_i \leq b$ (usually while optimizing some property that prevents us from setting all the x_i to zero). We'll assume that the a_i and b are all integers.

For #KNAPSACK, we want to compute $|S|$, where S is the set of all assignments to the x_i that make $\sum_{i=1}^{n} a_i x_i \leq b$.

There is a well-known **fully polynomial-time approximation scheme** for optimizing KNAPSACK, based on dynamic programming. The idea is that a maximum-weight solution can be found exactly in time polynomial in b, and if b is too large, we can reduce it by rescaling all the a_i and b at the cost of a small amount of error. A similar idea is used in Dyer's algorithm: the KNAPSACK problem is rescaled so that size of the solution set S' of the rescaled version can be computed in polynomial time. Sampling is then used to determine what proportion of the solutions in S' correspond to solutions of the original problem.

Scaling step: Let $a'_i = \lfloor n^2 a_i / b \rfloor$. Then $0 \leq a'_i \leq n^2$ for all i. Taking the floor creates some error: if we try to reconstruct a_i from a'_i, the best we can do is argue that $a'_i \leq n^2 a_i / b < a'_i + 1$ implies $(b/n^2) a'_i \leq a_i < (b/n^2) a'_i + (b/n^2)$. The reason for using n^2 as our rescaled bound is that the total error in the upper bound on a_i, summed over all i, is bounded by $n(b/n^2) = b/n$, a fact that will become important soon.

Let $S' = \{\vec{x} \mid \sum_{i=1}^{n} a'_i x_i \leq n^2\}$ be the set of solutions to the rescaled knapsack problem, where we substitute a'_i for a_i and $n^2 = (n^2/b)b$ for b.

Claim: $S \subseteq S'$. Proof: $\vec{x} \in S$ if and only if $\sum_{i=1}^{n} a_i x_i \leq b$. But then

$$\begin{aligned}
\sum_{i=1}^{n} a'_i x_i &= \sum_{i=1}^{n} \left\lfloor n^2 a_i / b \right\rfloor x_i \\
&\leq \sum_{i=1}^{n} (n^2/b) a_i x_i \\
&= (n^2/b) \sum_{i=1}^{n} a_i x_i \\
&\leq n^2,
\end{aligned}$$

which shows $\vec{x} \in S'$.

The converse does not hold. However, we can argue that any $\vec{x} \in S'$ can be shoehorned into S by setting at most one of the x_i to 0. Consider the set of all positions i such that $x_i = 1$ and $a_i > b/n$. If this set is empty, then $\sum_{i=1}^{n} a_i x_i \leq \sum_{i=1}^{n} b/n = b$, and $\vec{x}$ is already in S. Otherwise, pick any position i with $x_i = 1$ and $a_i > b/n$, and let $y_j = 0$ when $j = i$ and $y_j = x_j$ otherwise. Then

$$\begin{aligned}
\sum_{j=1}^{n} a_j y_j &= \sum_{j=1}^{n} a_j x_j - a_i \\
&< \sum_{j=1}^{n} ((b/n^2) a'_j + b/n^2) x_j - b/n \\
&\leq (b/n^2) \sum_{j=1}^{n} a'_j x_j + b/n - b/n \\
&\leq (b/n^2) n^2 \\
&= b.
\end{aligned}$$

Applying this mapping to all elements $\vec{x}$ of S maps at most $n+1$ of them to each $\vec{y}$ in S'; it follows that $|S'| \leq (n+1)|S|$, which means that if we can sample elements of S' uniformly, each sample will hit S with probability at least $1/(n+1)$.

To compute $|S'|$, let $C(k,m) = \left| \left\{ \vec{x} \;\middle|\; \sum_{i=1}^{k} a'_i x_i \leq m \right\} \right|$ be the number of subsets of $\{a'_1, \ldots, a'_k\}$ that sum to m or less. Then $C(k,m)$ satisfies the recurrence

$$\begin{aligned}
C(k,m) &= C(k-1, m-a'_k) + C(k-1, m) \\
C(0,m) &= 1
\end{aligned}$$

where k ranges from 0 to n and m ranges from 0 to n^2, and we treat $C(k-1, m-a'_k) = 0$ if $m - a'_k < 0$. The idea is that $C(k-1, m-a'_k)$ counts all the ways to make m if we include a'_k, and $C(k-1, m)$ counts all the ways to make m if we exclude it. The base case corresponds to the empty set (which sums to $\leq m$ no matter what m is).

We can compute a table of all values of $C(k, m)$ by iterating through m in increasing order; this takes $O(n^3)$ time. At the end of this process, we can read off $|S'| = C(n, n^2)$.

But we can do more than this: we can also use the table of counts to sample uniformly from S'. The probability that $x'_n = 1$ for a uniform random element of S' is exactly $C(n-1, n^2 - a'_n)/C(n, n^2)$; having chosen $x'_n = 1$ (say), the probability that $x'_{n-1} = 1$ is then $C(n-2, n^2 - a'_n - a'_{n-1})/C(n-2, n^2 - a'_n)$, and so on. So after making $O(n)$ random choices (with $O(1)$ arithmetic operations for each choice to compute the probabilities) we get a uniform element of S', which we can test for membership in S in an additional $O(n)$ operations.

We've already established that $|S|/|S'| \geq 1/(n+1)$, so we can apply Lemma 10.2.1 to get ϵ relative error with probability at least $1-\delta$ using $\frac{3(n+1)}{\epsilon^2} \ln \frac{2}{\delta}$ samples. This gives a cost of $O(n^2 \log(1/\delta)/\epsilon^2)$ for the sampling step, or a total cost of $O(n^3 + n^2 \log(1/\delta)/\epsilon^2)$ after including the cost of building the table (in practice, the second term will dominate unless we are willing to accept $\epsilon = \omega(1/\sqrt{n})$).

It is possible to improve this bound. Dyer [Dye03] shows that using **randomized rounding** on the a'_i instead of just truncating them gives a FPRAS that runs in $O(n^{5/2}\sqrt{\log(1/\epsilon)} + n^2/\epsilon^2)$ time.

10.5 Approximating exponentially improbable events

For #DNF and #KNAPSACK, we saw how restricting our sampling to a cleverly chosen sample space could boost the hit ratio ρ to something that gave a reasonable number of samples using Lemma 10.2.1. For other problems, it is often not clear how to do this directly, and the best sample spaces we can come up with make our target points an exponentially small fraction of the whole.

In these cases, it is sometimes possible to approximate this exponentially small fraction as a product of many more reasonable ratios. The idea is to express our target set as the last of a sequence of sets $S_0, S_1, \ldots, S_k$, where we can compute the size of S_0 and can estimate $|S_{i+1}|/|S_i|$ accurately for each i. This gives $|S_k| = |S_0| \cdot \prod_{i=1}^{k} \frac{|S_{i+1}|}{|S_i|}$, with a relative error that grows

roughly linearly with k. Specifically, if we can approximate each $|S_{i+1}|/|S_i|$ ratio to between $1-\epsilon$ and $1+\epsilon$ of the correct value, then the product of these ratios will be between $(1-\epsilon)^k$ and $(1+\epsilon)^k$ of the correct value; these bounds approach $1-\epsilon k$ and $1+\epsilon k$ in the limit as ϵk goes to zero, using the binomial theorem, although to get a real bound we will need to do more careful error analysis.

10.5.1 Matchings

We saw in §9.5.8.3 that a random walk on matchings on a graph with m edges has mixing time $\tau_2 \leq 8m^4$, where the walk is defined by selecting an edge uniformly at random and flipping whether it is in the matching or not, while rejecting any steps that produce a non-matching. This allows us to sample matchings of a graph with δ total variation distance from the uniform distribution in $O\left(m^4 \log m \log \frac{1}{\delta}\right)$ time.

Suppose now that we want to count matchings instead of sampling them. It's easy to show that for any particular edge $uv \in G$, at least half of all matchings in G don't include uv: the reason is that if M is a matching in G, then $M' = M \setminus \{uv\}$ is also a matching, and at most two matchings M' and $M' \cup \{uv\}$ are mapped to any one M' by this mapping.

Order the edges of G arbitrarily as $e_1, e_2, \ldots, e_m$. Let S_i be the set of matchings in $G \setminus \{e_1 \ldots e_i\}$. Then S_0 is the set of all matchings, and we've just argued that $\rho_{i+1} = |S_{i+1}|/|S_i| \geq 1/2$. We also know that $|S_m|$ counts the number of matchings in a graph with no edges, so it's exactly one. So we can use the product-of-ratios trick to compute $S_0 = \prod_{i=1}^{m} \frac{|S_i|}{|S_{i+1}|}$.

Using a random walk of length $O\left(8m^4 \log m \log \frac{1}{\eta}\right)$, we can sample matchings from S_i (which is just the set of matchings on a particular graph that happens to be our original graph minus some of its edges) so that our probability ρ' of getting a matching in S_{i+1} is between $(1-\eta)\rho_{i+1}$ and $(1+\eta)\rho_{i+1}$. From Lemma 10.2.1, we can estimate ρ' within relative error γ with probability at least $1-\zeta$ using $O\left(\frac{1}{\gamma^2\rho'} \log \frac{1}{\zeta}\right) = O\left(\frac{1}{\gamma} \log \frac{1}{\zeta}\right)$ samples. Combined with the error on ρ', this gives relative error at most $\gamma + \eta + \gamma\eta$ in $O\left(m^4 \log m \log \frac{1}{\eta} \log \frac{1}{\gamma} \log \frac{1}{\zeta}\right)$ operations.[1] If we then multiply out all the estimates for $|S_i|/|S_{i+1}|$, we get an estimate of S_0 that is at most $(1+\gamma+\eta+\gamma\eta)^m$ times the correct value with probability

[1] This is the point where sensible people start hauling out the $\tilde{O}$ notation, where a function is $\tilde{O}(f(n))$ if it $O(f(n)g)$ where g is polylogarithmic in n and any other parameters that may be running around ($\frac{1}{\epsilon}$, $\frac{1}{\eta}$, etc.).

at least $1 - m\zeta$ (with a similar bound on the other side), in total time $O\left(m^5 \log m \log \frac{1}{\eta} \log \frac{1}{\gamma} \log \frac{1}{\zeta}\right)$.

To turn this into a fully polynomial-time approximation scheme, given ϵ, δ, and m, we need to select η, γ, and ζ to get relative error ϵ with probability at least $1 - \delta$. Letting $\zeta = \delta/m$ gets the δ part. For ϵ, we need $(1 + \gamma + \eta + \gamma\eta)^m \leq 1 + \epsilon$. Suppose that $\epsilon < 1$ and let $\gamma = \eta = \epsilon/6m$. Then

$$\begin{aligned}(1 + \gamma + \eta + \gamma\eta)^m &\leq \left(1 + \frac{\epsilon}{2m}\right)^m \\ &\leq e^{\epsilon/2} \\ &\leq 1 + \epsilon.\end{aligned}$$

Plugging these values into our cost formula gives $O\left(m^5 \log m \log^2 \frac{6m}{\epsilon} \log \frac{m}{\delta}\right) = O\left(m^5 \log m \left(\log m + \log \frac{1}{\epsilon} + \log \frac{1}{\delta}\right)\right) = \tilde{O}(m^5)$ time.

10.5.2 Other applications

Similar methods work for other problems that self-reduce by restricting particular features of the solution. Examples include colorings (fix the color of some vertex), independent sets (remove a vertex), and approximating the volume of a convex body (take the intersection with a sphere of appropriate radius; see [MR95, §11.4] for a slightly less sketchy description). We will not go into details on any of these applications here.

Chapter 11

The probabilistic method

The **probabilistic method** is a tool for proving the existence of objects with particular combinatorial properties, by showing that some process generates these objects with nonzero probability.

The relevance of this to randomized algorithms is that in some cases we can make the probability large enough that we can actual produce such objects.

We'll mostly be following Chapter 5 of [MR95] with some updates for more recent results. If you'd like to read more about these technique, the standard reference on the probabilistic method in combinatorics is the text of Alon and Spencer [AS92].

11.1 Randomized constructions and existence proofs

Suppose we want to show that some object exists, but we don't know how to construct it explicitly. One way to do this is to devise some random process for generating objects, and show that the probability that it generates the object we want is greater than zero. This implies that something we want exists, because otherwise it would be impossible to generate; and it works even if the nonzero probability is very, very small. The systematic development of the method is generally attributed to the notoriously productive mathematician Paul Erdős and his frequent collaborator Alfréd Rényi.

From an algorithmic perspective, the probabilistic method is useful mainly when we can make the nonzero probability substantially larger than zero—and especially if we can recognize when we've won. But sometimes just demonstrating the existence of an object is a start.

We give a couple of example of the probabilistic method in action below.

In each case, the probability that we get a good outcome is actually pretty high, so we could in principle generate a good outcome by retrying our random process until it works. There are some more complicated examples of the method for which this doesn't work, either because the probability of success is vanishingly small, or because we can't efficiently test whether what we did succeeded (the last example below may fall into this category). This means that we often end up with objects whose existence we can demonstrate even though we can't actually point to any examples of them. For example, it is known that there exist **sorting networks** (a special class of circuits for sorting numbers in parallel) that sort in time $O(\log n)$, where n is the number of values being sorted [AKS83]; these can be generated randomly with nonzero probability. But the best explicit constructions of such networks take time $\Theta(\log^2 n)$, and the question of how to find an *explicit* network that achieves $O(\log n)$ time has been open for over 25 years now despite many efforts to solve it.

11.1.1 Unique hats

A collection of n robots each wishes to own a unique hat. Unfortunately, the State Ministry for Hat Production only supplies one kind of hat. A robot will only be happy if (a) it has a hat, and (b) no robot it sees has a hat. Fortunately each robot can only see k other robots. How many robots can be happy?

We could try to be clever about answering this question, or we could apply the probabilistic method. Suppose we give each robot a hat with independent probability p. Then the probability that any particular robot r is happy is pq^k, where $q = 1 - p$ is the probability that a robot doesn't have a hat and q^k gives the probability that none of the robots that r sees has a hat. If we let X_r be the indicator for the event that r is happy, we have $\mathrm{E}[X_r] = pq^k$ and the expected number of happy robots is $\mathrm{E}[\sum X_r] = \sum \mathrm{E}[X_r] = npq^k$. Since we can achieve this value on average, there must be some specific assignment that achieves it as well.

To choose a good p, we apply the usual calculus trick of looking for a maximum by looking for the place where the derivative is zero. We have $\frac{d}{dp}npq^k = nq^k - nkpq^{k-1}$ (since $\frac{dq}{dp} = -1$), so we get $nq^k - nkpq^{k-1} = 0$. Factoring out n and q^{k-1} gives $q - pk = 0$ or $1 - p - pk = 0$ or $p = 1/(k+1)$. For this value of p, the expected number of happy robots is exactly $n\left(\frac{1}{k+1}\right)\left(1 - \frac{1}{k+1}\right)^k$. For large k the last factor approaches $1/e$, giving us approximately $(1/e)\frac{n}{k+1}$ happy robots.

Up to constant factors this is about as good an outcome as we can hope for: we can set things up so that our robots are arranged in groups of $k+1$, where each robot sees all the other robots in its group, and here we can clearly only have 1 happy robot per group, for a maximum of $n/(k+1)$ happy robots.

Note that we can improve the constant $1/e$ slightly by being smarter than just handing out hats at random. Starting with our original n robots, look for a robot that is observed by the smallest number of other robots. Since there are only nk pairs of robots (r_1, r_2) where r_1 sees r_2, one of the n robots is only seen by at most k other robots. Now give this robot a hat, and give no hat to any robot that sees it or that it sees. This produces 1 happy robot while knocking out at most $2k+1$ robots total. Now remove these robots from consideration and repeat the analysis on the remaining robots, to get another happy robot while again knocking out at most $2k+1$ robots. Iterating this procedure until no robots are left gives at least $n/(2k+1)$ happy robots; this is close to $(1/2)\frac{n}{k+1}$ for large k, which is a bit better than $(1/e)\frac{n}{k+1}$. (It may be possible to improve the constant further with more analysis.) This shows that the probabilistic method doesn't necessarily produce better results than we can get by being smart. But the hope is that with the probabilistic method we don't have to be smart, and for some problems it seems that solving them without using the probabilistic method requires being exceptionally smart.

11.1.2 Ramsey numbers

Consider a collection of n schoolchildren, and imagine that each pair of schoolchildren either like each other or hate each other. We assume that these preferences are symmetric: if x likes y, then y likes x, and similarly if x hates y, y hates x. Let $R(k,h)$ be the smallest value for n that ensures that among any group of n schoolchildren, there is either a subset of k children that all like each other or a subset of h children that all hate each other.[1]

It is not hard to show that $R(k,h)$ is finite for all k and h.[2] The exact

[1] In terms of graphs, any graph G with at least $R(k,h)$ nodes contains either a **clique** of size k or an **independent set** of size h.

[2] A simple proof, due to Erdős and Szekeres [ES35], proceeds by showing that $R(k,h) \leq R(k-1,h) + R(k,h-1)$. Given a graph G of size at least $R(k-1,h) + R(k,h-1)$, choose a vertex v and partition the graph into two induced subgraphs G_1 and G_2, where G_1 contains all the vertices adjacent to v and G_2 contains all the vertices not adjacent to v. Either $|G_1| \geq R(k-1,h)$ or $|G_2| \geq R(k,h-1)$. If $|G_1| \geq R(k-1,h)$, then G_1 contains either a clique of size $k-1$ (which makes a clique of size k in G when we add v to it) or an independent set of size h (which is also in G). Alternatively, if

value of $R(k,h)$ is known only for small values of k and h.[3] But we can use the probabilistic method to show that for $k = h$, it is reasonably large. The following theorem is due to Erdős, and was the first known lower bound on $R(k,k)$.

Theorem 11.1.1 ([Erd47])**.** *If $k \geq 3$, then $R(k,k) > 2^{k/2}$.*

Proof. Suppose each pair of schoolchildren flip a fair coin to decide whether they like each other or not. Then the probability that any particular set of k schoolchildren all like each other is $2^{-\binom{k}{2}}$ and the probability that they all hate each other is the same. Summing over both possibilities and all subsets gives a bound of $\binom{n}{k}2^{1-\binom{k}{2}}$ on the probability that there is at least one subset in which everybody likes or everybody hates everybody. For $n = 2^{k/2}$, we have

$$\begin{aligned}\binom{n}{k}2^{1-\binom{k}{2}} &\leq \frac{n^k}{k!}2^{1-\binom{k}{2}} \\ &= \frac{2^{k^2/2+1-k(k-1)/2}}{k!} \\ &= \frac{2^{k^2/2+1-k^2/2+k/2}}{k!} \\ &= \frac{2^{1+k/2}}{k!} \\ &< 1.\end{aligned}$$

Because the probability that there is an all-liking or all-hating subset is less than 1, there must be some chance that we get a collection that doesn't have one. So such a collection exists. It follows that $R(k,k) > 2^{k/2}$, because we have shown that not all collections at $n = 2^{k/2}$ have the Ramsey property. □

$|G_2| \geq R(k,h-1)$, then G_2 either gives us a clique of size k by itself or an independent set of size h after adding v. Together with the fact that $R(1,h) = R(k,1) = 1$, this recurrence gives $R(k,h) \leq \binom{(k-1)+(h-1)}{k-1}$.

[3] There is a fairly current table at `http://en.wikipedia.org/wiki/Ramsey's_Theorem`. Some noteworthy values are $R(3,3) = 6$, $R(4,4) = 18$, and $43 \leq R(5,5) \leq 49$. One problem with computing exact values is that as $R(k,h)$ grows, the number of graphs one needs to consider gets very big. There are $2^{\binom{n}{2}}$ graphs with n vertices, and even detecting the presence or absence of a moderately-large clique or independent set in such a graph can be expensive. This pretty much rules out any sort of brute-force approach based on simply enumerating candidate graphs.

The last step in the proof uses the fact that $2^{1+k/2} < k!$ for $k \geq 3$, which can be tested explicitly for $k = 3$ and proved by induction for larger k. The resulting bound is a little bit weaker than just saying that n must be large enough that $\binom{n}{k} 2^{1-\binom{k}{2}} \geq 1$, but it's easier to use.

The proof can be generalized to the case where $k \neq h$ by tweaking the bounds and probabilities appropriately. Note that even though this process generates a graph with no large cliques or independent sets with reasonably high probability, we don't have any good way of testing the result, since testing for the existence of a clique is **NP**-hard.

11.1.3 Directed cycles in tournaments

In the previous example we used the probabilistic method to show that there existed a structure that didn't contain some particular substructure. For this example we will do the reverse: show that there exists a structure that contains many instances of a particular substructure.

Imagine that n wrestlers wrestle in a round-robin tournament, so that ever wrestler wrestles every other wrestler exactly once, and so that for each pair of wrestlers one wrestler beats the other. Consider a cycle of wrestlers $x_1, x_2, \ldots, x_n$ such that each wrestler x_i beats x_{i+1} and x_n beats x_1. (Note that we consider two cycles equivalent if one is a cyclic shift of the other.)

Theorem 11.1.2. *For $n \geq 3$, there exists a tournament with at least $(n-1)!2^{-n}$ directed cycles.*

Proof. Technical detail: We need $n \geq 3$ because the edges in a 2-cycle always go in opposite directions.

Flip a coin to decide the outcome of each pairing. For any particular sequence $x_1, \ldots, x_n$, the probability that it is a directed cycle is then 2^{-n} (since there are n edges in the sequence and each has independent probability 2^{-1} of going the right way). Now let σ range over all $(n-1)!$ cycles and define X_σ as the indicator variable for the event that each edge is directed the right way. We've just shown that $\mathrm{E}[X_\sigma] = 2^{-n}$; so the expected total number of cycles is $\mathrm{E}[\sum_\sigma X_\sigma] = \sum_\sigma \mathrm{E}[X_\sigma] = (n-1)!2^{-n}$. But if the average value is at least $(n-1)!2^{-n}$, there must be some specific outcome that gives a value at least this high. □

11.2 Approximation algorithms

One use of a randomized construction is to approximate the solution to an otherwise difficult problem. In this section, we start with a trivial

approximation algorithm for the largest cut in a graph, and then show a more powerful randomized approximation algorithm, due to Goemans and Williamson [GW94], that gets a better approximation ratio for a much larger class of problems.

11.2.1 MAX CUT

We've previously seen (§2.3.1.2) a randomized algorithm for finding small cuts in a graph. What if we want to find a large cut?

Here is a particularly brainless algorithm that finds a large cut. For each vertex, flip a coin: if the coin comes up heads, put the vertex in S, otherwise, put in it T. For each edge, there is a probability of exactly $1/2$ that it is included in the S–T cut. It follows that the expected size of the cut is exactly $m/2$.

One consequence of this is that every graph has a global cut that includes at least half the edges. Another is that this algorithm finds such a cut, with probability at least $\frac{1}{m+1}$. To prove this, let X be the random variable representing the number of edges include in the cut, and let p be the probability that $X \geq m/2$. Then

$$\begin{aligned} m/2 &= \mathrm{E}\left[X\right] \\ &= (1-p)\,\mathrm{E}\left[X \mid X < m/2\right] + p\,\mathrm{E}\left[X \mid X \geq m/2\right] \\ &\leq (1-p)\frac{m-1}{2} + pm. \end{aligned}$$

Solving this for p gives the claimed bound.[4]

By running this enough times to get a good cut, we get a polynomial-time randomized algorithm for approximating the **maximum cut** within a factor of 2, which is pretty good considering that MAX CUT is **NP**-hard.

There exist better approximation algorithms. Goemans and Williamson [GW95] give a 0.87856-approximation algorithm for MAX CUT based on randomized rounding of a semidefinite program. We won't attempt to present this here, but we will describe (in §11.2.2) an earlier result, also due to Goemans and Williamson [GW94], that gives a $\frac{3}{4}$-approximation to MAX SAT using a similar technique.

[4] This is tight for $m = 1$, but I suspect it's an underestimate for larger m. The main source of slop in the analysis seems to be the step $\mathrm{E}\left[X \mid X \geq m/2\right] \leq m$; using a concentration bound, we should be able to show a much stronger inequality here and thus a much larger lower bound on p.

11.2.2 MAX SAT

Like MAX CUT, MAX SAT is an **NP**-hard optimization problem that sounds like a very short story about Max. We are given a **satisfiability problem** in **conjunctive normal form**: as a set of m **clauses**, each of which is the OR of a bunch of variables or their negations. We want to choose values for the n variables that **satisfy** as many of the clauses as possible, where a clause is satisfied if it contains a true variable or the negation of a false variable.[5]

We can instantly satisfy at least $m/2$ clauses on average by assigning values to the variables independently and uniformly at random; the analysis is the same as in §11.2.1 for large cuts, since a random assignment makes the first variable in a clause true with probability 1/2. Since this approach doesn't require thinking and doesn't use the fact that many of our clauses may have more than one variable, we can probably do better. Except we can't do better in the worst case, because it might be that our clauses consist entirely of x and $\neg x$ for some variable x; clearly, we can only satisfy half of these. We could do better if we knew all of our clauses consisted of at least k distinct literals (satisfied with probability $1 - 2^{-k}$), but we can't count on this. We also can't count on clauses not being duplicated, so it may turn out that skipping a few hard-to-satisfy small clauses hurts us if they occur many times.

Our goal will be to get a good **approximation ratio**, defined as the ratio between the number of clauses we manage to satisfy and the actual maximum that can be satisfied. The tricky part in designing a **approximation algorithms** is showing that the denominator here won't be too big. We can do this using a standard trick, of expressing our original problem as an **integer program** and then **relaxing** it to a **linear program**[6] whose

[5] The presentation here follows [MR95, §5.2], which in turn is mostly based on a classic paper of Goemans and Williamson [GW94].

[6] A **linear program** is an optimization problem where we want to maximize (or minimize) some linear **objective function** of the variables subject to linear-inequality constraints. A simple example would be to maximize $x + y$ subject to $2x + y \leq 1$ and $x + 3y \leq 1$; here the **optimal solution** is the assignment $x = \frac{2}{5}, y = \frac{1}{5}$, which sets the objective function to its maximum possible value $\frac{3}{5}$. An **integer program** is a linear program where some of the variables are restricted to be integers. Determining if an integer program even has a solution is **NP**-complete; in contrast, linear programs can be solved in polynomial time. We can **relax** an integer program to a linear program by dropping the requirements that variables be integers; this will let us find a **fractional solution** that is at least as good as the best integer solution, but might be undesirable because it tells us to do something that is ludicrous in the context of our original problem, like only putting half a passenger on the next plane.

solution doesn't have to consist of integers. We then convert the fractional solution back to an integer solution by rounding some of the variables to integer values randomly in a way that preserves their expectations, a technique known as **randomized rounding**.[7]

Here is the integer program (taken from [MR95, §5.2]). We let $z_j \in \{0, 1\}$ represent whether clause C_j is satisfied, and let $y_i \in \{0, 1\}$ be the value of variable x_i. We also let C_j^+ and C_j^- be the set of variables that appear in C_j with and without negation. The problem is to maximize

$$\sum_{j=1}^{m} z_j$$

subject to

$$\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j,$$

for all j.

The main trick here is to encode OR in the constraints; there is no requirement that z_j is the OR of the y_i and $(1 - y_i)$ values, but we maximize the objective function by setting it that way.

Sadly, solving integer programs like the above is **NP**-hard (which is not surprising, since if we could solve this particular one, we could solve SAT). But if we drop the requirements that $y_i, z_j \in \{0, 1\}$ and replace them with $0 \leq y_i \leq 1$ and $0 \leq z_j \leq 1$, we get a linear program—solvable in polynomial time—with an optimal value at least as good as the value for the integer program, for the simple reason that any solution to the integer program is also a solution to the linear program.

The problem now is that the solution to the linear program is likely to be **fractional**: instead of getting useful 0–1 values, we might find out we are supposed to make x_i only 2/3 true. So we need one more trick to turn

Linear programming has an interesting history. The basic ideas were developed independently by Leonid Kantorovich in the Soviet Union and George Dantzig in the United States around the start of the Second World War. Kantorovich's work had direct relevance to Soviet planning problems, but wasn't pursued seriously because it threatened the political status of the planners, required computational resources that weren't available at the time, and looked suspiciously like trying to sneak a capitalist-style price system into the planning process; for a fictionalized account of this tragedy, see [Spu12]. Dantzig's work, which included the development of the **simplex method** for solving linear programs, had a higher impact, although its publication was delayed until 1947 by wartime secrecy.

[7] Randomized rounding was invented by Raghavan and Thompson [RT87]; the particular application here is due to Goemans and Williamson [GW94].

the fractional values back into integers. This is the randomized rounding step: given a fractional assignment $\hat{y}_i$, we set x_i to true with probability $\hat{y}_i$.

So what does randomized rounding do to clauses? In our fractional solution, a clause might have value $\hat{z}_j$, obtained by summing up bits and pieces of partially-true variables. We'd like to argue that the rounded version gives a similar probability that C_j is satisfied.

Suppose C_j has k variables; to make things simpler, we'll pretend that C_j is exactly $x_1 \vee x_2 \vee \ldots x_k$. Then the probability that C_j is satisfied is exactly $1 - \prod_{i=1}^{k}(1-\hat{y}_i)$. This quantity is minimized subject to $\sum_{i=1}^{k} \hat{y}_i \geq \hat{z}_j$ by setting all $\hat{y}_i$ equal to $\hat{z}_j/k$ (easy application of Lagrange multipliers, or can be shown using a convexity argument). Writing z for $\hat{z}_j$, this gives

$$\begin{aligned}
\Pr\left[C_j \text{ is satisfied}\right] &= 1 - \prod_{i=1}^{k}(1-\hat{y}_i) \\
&\geq 1 - \prod_{i=1}^{k}(1-z/k) \\
&= 1 - (1-z/k)^k \\
&\geq z(1-(1-1/k)^k). \\
&\geq z(1-1/e).
\end{aligned}$$

The second-to-last step looks like a typo, but it actually works. The idea is to observe that the function $f(z) = 1-(1-z/k)^k$ is concave (Proof: $\frac{d^2}{dz^2}f(z) = -\frac{k-1}{k}(1-z/k)^{k-2} < 0$), while $g(z) = z(1-(1-1/k)^k)$ is linear, so since $f(0) = 0 = g(0)$ and $f(1) = 1-(1-1/k)^k = g(1)$, any point in between must have $f(z) \geq g(z)$.

Since each clause is satisfied with probability at least $\hat{z}_j(1-1/e)$, the expected number of satisfied clauses is at least $(1-1/e)\sum_j \hat{z}_j$, which is at least $(1-1/e)$ times the optimum. This gives an approximation ratio of slightly more than 0.632, which is better than 1/2, but still kind of weak.

So now we apply the second trick from [GW94]: we'll observe that, on a per-clause basis, we have a randomized rounding algorithm that is good at satisfying small clauses (the coefficient $(1-(1-1/k)^k)$ goes all the way up to 1 when $k = 1$), and our earlier dumb algorithm that is good at satisfying big clauses. We can't combine these directly (the two algorithms demand different assignments), but we can run both in parallel and take whichever gives a better result.

To show that this works, let X_j be indicator for the event that clause j is satisfied by the randomized-rounding algorithm and Y_j the indicator for the

even that it is satisfied by the simple algorithm. Then if C_j has k literals,

$$\begin{aligned}\mathrm{E}[X_j] + \mathrm{E}[Y_j] &\geq (1 - 2^{-k}) + (1 - (1 - 1/k)^k)\hat{z}_j \\ &\geq ((1 - 2^{-k}) + (1 - (1 - 1/k)^k))\hat{z}_j \\ &= (2 - 2^{-k} - (1 - 1/k)^k)\hat{z}_j.\end{aligned}$$

The coefficient here is exactly $3/2$ when $k = 1$ or $k = 2$, and rises thereafter, so for integer k we have $\mathrm{E}[X_j] + \mathrm{E}[Y_j] \geq (3/2)\hat{z}_j$. Summing over all j then gives $\mathrm{E}\left[\sum_j X_j\right] + \mathrm{E}\left[\sum_j Y_j\right] \geq (3/2)\sum_j \hat{z}_j$. But then one of the two expected sums must beat $(3/4)\sum_j \hat{z}_j$, giving us a (3/4)-approximation algorithm.

11.3 The Lovász Local Lemma

Suppose we have a finite set of bad events $\mathcal{A}$, and we want to show that with nonzero probability, none of these events occur. Formally, we want to show $\Pr\left[\bigcap_{A\in\mathcal{A}} \bar{A}\right] > 0$.

Our usual trick so far has been to use the union bound (4.1.1) to show that $\sum_{A\in\mathcal{A}} \Pr[A] < 1$. But this only works if the events are actually improbable. If the union bound doesn't work, we might be lucky enough to have the events be independent; in this case, $\Pr\left[\bigcap_{A\in\mathcal{A}} \bar{A}\right] = \prod_{A\in\mathcal{A}} \Pr\left[\bar{A}\right] > 0$, as long as each event $\bar{A}$ occurs with positive probability. But most of the time, we'll find that the events we care about aren't independent, so this won't work either.

The **Lovász Local Lemma** [EL75] handles a situation intermediate between these two extremes, where events are generally not independent of each other, but each collection of events that are not independent of some particular event A has low total probability. In the original version; it's non-constructive: the lemma shows a nonzero probability that none of the events occur, but this probability may be very small if we try to sample the events at random and there is no guidance for how to find a particular outcome that makes all the events false.

Subsequent work [Bec91, Alo91, MR98, CS00, Sri08, Mos09, MT10] showed how, when the events A are determined by some underlying set of independent variables and independence between two events is detected by having non-overlapping sets of underlying variables, an actual solution could be found in polynomial expected time. The final result in this series, due to Moser and Tardos [MT10], gives the same bounds as in the original non-constructive lemma, using the simplest algorithm imaginable: whenever

some bad event A occurs, squish it by resampling all of its variables, and continue until no bad events are left.

11.3.1 General version

A formal statement of the general lemma is:[8]

Lemma 11.3.1. *Let $\mathcal{A} = A_1, \ldots, A_m$ be a finite collection of events on some probability space, and for each $A \in \mathcal{A}$, let $\Gamma(A)$ be a set of events such that A is independent of all events not in $\Gamma^+ A = \{A\} \cup \Gamma(A)$. If there exist real numbers $0 < x_A < 1$ such that, for all events $A \in \mathcal{A}$*

$$\Pr[A] \leq x_A \prod_{B \in \Gamma(A)} (1 - x_B), \tag{11.3.1}$$

then

$$\Pr\left[\bigcap_{A \in \mathcal{A}} \bar{A}\right] \geq \prod_{A \in \mathcal{A}} (1 - x_A). \tag{11.3.2}$$

In particular, this means that the probability that none of the A_i occur is not zero, since we have assumed $x_{A_i} < 1$ holds for all i.

The role of x_A in the original proof is to act as an upper bound on the probability that A occurs given that some collection of other events doesn't occur. For the constructive proof, the x_A are used to show a bound on the number of resampling steps needed until none of the A occur.

11.3.2 Symmetric version

For many applications, having to come up with the x_A values can be awkward. The following symmetric version is often used instead:

Corollary 11.3.2. *Let $\mathcal{A}$ and Γ be as in Lemma 11.3.1. Suppose that there are constants p and d, such that for all $A \in \mathcal{A}$, we have $\Pr[A] \leq p$ and $|\Gamma(A)| \leq d$. Then if $ep(d+1) < 1$, $\Pr\left[\bigcap_{A \in \mathcal{A}} \bar{A}\right] \neq 0$.*

Proof. Basically, we are going to pick a single value x such that $x_A = x$ for all A in $\mathcal{A}$, and (11.3.1) is satisfied. This works as long as $p \leq x(1-x)^d$, as in this case we have, for all A, $\Pr[A] \leq p \leq x(1-x)^d \leq x(1-x)^{|\Gamma(A)|} = x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B)\right)$.

[8] This version is adapted from [MT10].

For fixed d, $x(1-x)^d$ is maximized using the usual trick: $\frac{d}{dx}x(1-x)^d = (1-x)^d - xd(1-x)^{d-1} = 0$ gives $(1-x) - xd = 0$ or $x = \frac{1}{d+1}$. So now we need $p \leq \frac{1}{d+1}\left(1 - \frac{1}{d+1}\right)^d$. It is possible to show that $1/e < \left(1 - \frac{1}{d+1}\right)^d$ for all $d \geq 0$ (see the solution to Problem C.1.4). So $ep(d+1) \leq 1$ implies $p \leq \frac{1}{e(d+1)} \leq \left(1 - \frac{1}{d+1}\right)^d \frac{1}{d+1} \leq x(1-x)^{|\Gamma(A)|}$ as required by Lemma 11.3.1. □

11.3.3 Applications

11.3.3.1 Graph coloring

Let's start with a simple application of the local lemma where we know what the right answer should be. Suppose we want to color the vertices of a cycle with c colors, so that no edge has two endpoints with the same color. How many colors do we need?

Using brains, we can quickly figure out that $c = 3$ is enough. Without brains, we could try coloring the vertices randomly: but in a cycle with n vertices and n edges, on average n/c of the edges will be monochromatic, since each edge is monochromatic with probability $1/c$. If these bad events were independent, we could argue that there was a $(1-1/c)^n > 0$ probability that none of them occurred, but they aren't, so we can't. Instead, we'll use the local lemma.

The set of bad events $\mathcal{A}$ is just the set of events $A_i = [\text{edge } i \text{ is monochromatic}]$. We've already computed $p = 1/c$. To get d, notice that each edge only shares a vertex with two other edges, so $|\Gamma(A_i)| \leq 2$. Corollary 11.3.2 then says that there is a good coloring as long as $ep(d+1) = 3e/c \leq 1$, which holds as long as $c \geq 9$. We've just shown we can 9-color a cycle. If we look more closely at the proof of Corollary 11.3.2, we can see that $p \leq \frac{1}{3}\left(1 - \frac{1}{3}\right)^2 = \frac{4}{27}$ would also work; this says we can 7-color a cycle. Still not as good as what we can do if we are paying attention, but not bad for a procedure that doesn't use the structure of the problem much.

11.3.3.2 Satisfiability of k-CNF formulas

A more sophisticated application is demonstrating satisfiability for k-CNF formulas where each variable appears in a bounded number of clauses. Recall that a **k-CNF formula** consists of m **clauses**, each of which consists of exactly k variables or their negations (collectively called **literals**. It is **satisfied** if at least one literal in every clause is assigned the value true.

Suppose that each variable appears in at most ℓ clauses. Let $\mathcal{A}$ consist of

all the events $A_i = [\text{clause } i \text{ is not satisfied}]$. Then, for all i, $\Pr[A_i] = 2^{-k}$ exactly, and $|\Gamma(A_i)| \leq d = k(\ell - 1)$ since each variable in A_i is shared with at most $\ell - 1$ other clauses, and A_i will be independent of all events A_j with which it doesn't share a variable. So if $ep(d+1) = e2^{-k}(k(\ell-1)+1) \leq 1$, which holds if $\ell \leq \frac{2^k}{ek} + 1$, Corollary 11.3.2 tells us that a satisfying assignment exists.[9]

Corollary 11.3.2 doesn't let us actually find a satisfying assignment, but it turns out we can do that too. We'll return to this when we talk about the constructive version of the local lemma in §11.3.5

11.3.4 Non-constructive proof

This is essentially the same argument presented in [MR95, §5.5], but we adapt the notation to match the statement in terms of neighborhoods $\Gamma(A)$ instead of edges in a **dependency graph**.[10]

We show by induction on $|S|$ that for any A and any $S \subseteq \mathcal{A}$ with $A \notin S$,

$$\Pr\left[A \,\middle|\, \bigcap_{B \in S} \bar{B}\right] \leq x_A. \tag{11.3.3}$$

When $|S| = 0$, this just says $\Pr[A] \leq x_A$, which follows immediately from (11.3.1).

For larger S, split S into $S_1 = S \cap \Gamma(A)$, the events in S that might not be independent of A; and $S_2 = S \setminus \Gamma(A)$, the events in S that we know to be independent of A. If $S_2 = S$, then A is independent of all events in S, and (11.3.3) follows immediately from $\Pr\left[A \,\middle|\, \bigcap_{B \in S} \bar{B}\right] = \Pr[A] \leq x_A \prod_{B \in \Gamma(A)}(1 - x_B) \leq x_A$. Otherwise $|S_2| < |S|$, which means that we can assume the induction hypothesis holds for S_2.

[9] To avoid over-selling this claim, it's worth noting that the bound on ℓ only reaches 2 at $k = 4$, although it starts growing pretty fast after that.

[10] The two formulations are identical, since we can always represent the neighborhoods $\Gamma(A)$ by creating an edge from A to B when B is in $\Gamma(A)$; and conversely we can convert a dependency graph into a neighborhood structure by making $\Gamma(A)$ the set of successors of A in the dependency graph.

Write C_1 for the event $\bigcap_{B \in S_1} \bar{B}$ and C_2 for the event $\bigcap_{B \in S_2} \bar{B}$. Then

$$\begin{aligned}
\Pr\left[A \,\middle|\, \bigcap_{B \in S} \bar{B}\right] &= \frac{\Pr[A \cap C_1 \cap C_2]}{\Pr[C_1 \cap C_2]} \\
&= \frac{\Pr[A \cap C_1 \mid C_2] \Pr[C_2]}{\Pr[C_1 \mid C_2] \Pr[C_2]} \\
&= \frac{\Pr[A \cap C_1 \mid C_2]}{\Pr[C_1 \mid C_2]}. \qquad (11.3.4)
\end{aligned}$$

We don't need to do anything particularly clever with the numerator:

$$\begin{aligned}
\Pr[A \cap C_1 \mid C_2] &\leq \Pr[A \mid C_2] \\
&= \Pr[A] \\
&\leq x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B) \right), \qquad (11.3.5)
\end{aligned}$$

from (11.3.1) and the fact that A is independent of all B in S_2 and thus of C_2.

For the denominator, we expand C_1 back out to $\bigcap_{B \in S_1} \bar{B}$ and break out the induction hypothesis. To bound $\Pr\left[\bigcap_{B \in S_1} \bar{B} \,\middle|\, C_2\right]$, we order S_1 arbitrarily as $\{B_1, \ldots, B_r\}$, for some r, and show by induction on ℓ as ℓ goes from 1 to r that

$$\Pr\left[\bigcap_{i=1}^{\ell} \bar{B}_i \,\middle|\, C_2\right] \geq \prod_{i=1}^{\ell} (1 - x_{B_i}). \qquad (11.3.6)$$

The proof is that, for $\ell = 1$,

$$\begin{aligned}
\Pr\left[\bar{B}_1 \,\middle|\, C_2\right] &= 1 - \Pr[B_1 \mid C_2] \\
&\geq 1 - x_{B_1}
\end{aligned}$$

using the outer induction hypothesis (11.3.3), and for larger ℓ, we can compute

$$\begin{aligned}
\Pr\left[\bigcap_{i=1}^{\ell} \bar{B}_i \,\middle|\, C_2\right] &= \Pr\left[\bar{B}_\ell \,\middle|\, \left(\bigcap_{i=1}^{\ell-1} \bar{B}_i\right) \cap C_2\right] \cdot \Pr\left[\bigcap_{i=1}^{\ell-1} \bar{B}_i \,\middle|\, C_2\right] \\
&\geq (1 - x_{B_\ell}) \prod_{i=1}^{\ell-1} (1 - x_{B_i}) \\
&= \prod_{i=1}^{\ell} (1 - x_{B_i}),
\end{aligned}$$

where the second-to-last step uses the outer induction hypothesis (11.3.3) for the first term and the inner induction hypothesis (11.3.6) for the rest. This completes the proof of the inner induction.

When $\ell = r$, we get

$$\begin{aligned}\Pr[C_1 \mid C_2] &= \Pr\left[\bigcap_{i=1}^{r} \bar{B}_i \,\middle|\, C_2\right] \\ &\geq \prod_{B \in S_1} (1 - x_B). \end{aligned} \tag{11.3.7}$$

Substituting (11.3.5) and (11.3.7) into (11.3.4) gives

$$\begin{aligned}\Pr\left[A \,\middle|\, \bigcap_{B \in S} \bar{B}\right] &\leq \frac{x_A \left(\prod_{B \in \Gamma(A)} (1 - x_B)\right),}{\prod_{B \in S_1} (1 - x_B)} \\ &= x_A \left(\prod_{B \in \Gamma(A) \setminus S_1} (1 - x_B)\right) \\ &\leq x_A.\end{aligned}$$

This completes the proof of the outer induction.

To get the bound (11.3.2), we reach back inside the proof and repeat the argument for (11.3.7) with $\bigcap_{A \in \mathcal{A}} \bar{A}$ in place of C_1 and without the conditioning on C_2. We order $\mathcal{A}$ arbitrarily as $\{A_1, A_2, \ldots, A_m\}$ and show by induction on k that

$$\Pr\left[\bigcap_{i=1}^{k} \bar{A}_i\right] \geq \prod_{i=1}^{k} (1 - x_{A_k}). \tag{11.3.8}$$

For the base case we have $k = 0$ and $\Pr[\Omega] \geq 1$, using the usual conventions on empty products. For larger k, we have

$$\begin{aligned}\Pr\left[\bigcap_{i=1}^{k} \bar{A}_i\right] &= \Pr\left[\bar{A}_k\right] \bigcap_{i=1}^{k-1} \bar{A}_i \\ &\geq (1 - x_{A_k}) \prod_{i=1}^{k-1} (1 - x_{A_i}) \\ &\geq \prod_{i=1}^{k} (1 - x_{A_k}),\end{aligned}$$

where in the second-to-last step we use (11.3.3) for the first term and (11.3.8) for the big product.

Setting $k = n$ finishes the proof.

11.3.5 Constructive proof

We now describe he constructive proof of the Lovász local lemma due to Moser and Tardos [MT10], which is based on a slightly more specialized construction of Moser alone [Mos09]. This version applies when our set of bad events $\mathcal{A}$ is defined in terms of a set of *independent* variables $\mathcal{P}$, where each $A \in \mathcal{A}$ is determined by some set of variables $\mathrm{vbl}(A) \subseteq \mathcal{P}$, and $\Gamma(A)$ is defined to be the set of all events $B \neq A$ that share at least one variable with A; i.e., $\Gamma(A) = \{B \in \mathcal{A} - \{A\} \mid \mathrm{vbl}(B) \cap \mathrm{vbl}(A) \neq \emptyset\}$.

In this case, we can attempt to find an assignment to the variables that makes none of the A occur using the obvious algorithm of sampling an initial state randomly, then resampling all variables in $\mathrm{vbl}(A)$ whenever we see some bad A occur. Astonishingly, this actually works in a reasonable amount of time, without even any cleverness in deciding which A to resample at each step, if the conditions for Lemma 11.3.1 hold for x that are not too large. In particular, we will show that a good assignment is found after each A is resampled at most $\frac{x_A}{1-x_A}$ times on average.

Lemma 11.3.3. *Under the conditions of Lemma 11.3.1, the Moser-Tardos procedure does at most*

$$\sum_A \frac{x_A}{1 - x_A}$$

resampling steps on average.

We defer the proof of Lemma 11.3.3 for the moment. For most applications, the following symmetric version is easier to work with:

Corollary 11.3.4. *Under the conditions of Corollary 11.3.2, the Moser-Tardos procedure does at most m/d resampling steps on average.*

Proof. Follows from Lemma 11.3.3 and the choice of $x_A = \frac{1}{d+1}$ in the proof of Corollary 11.3.2. □

How this expected m/d bound translates into actual time depends on the cost of each resampling step. The expensive part at each step is likely to be the cost of finding an A that occurs and thus needs to be resampled.

Intuitively, we might expect the resampling to work because if each $A \in \mathcal{A}$ has a small enough neighborhood $\Gamma(A)$ and a low enough probability, then whenever we resample A's variables, it's likely that we fix A and unlikely that we break too many B events in A's neighborhood. It turns out to be tricky to quantify how this process propagates outward, so the actual proof

uses a different idea that essentially looks at this process in reverse, looking for each resampled event A at a set of previous events whose resampling we can blame for making A occur, and then showing that this tree (which will include every resampling operation as one of its vertices) can't be too big.

The first step is to fix some strategy for choosing which event A to resample at each step. We don't care what this strategy is; we just want it to be the case that the sequence of events depends only on the random choices made by the algorithm in its initial sampling and each resampling. We can then define an **execution log** C that lists the sequence of events $C_1, C_2, C_3, \ldots$ that are resampled at each step of the execution.

From C we now construct a **witness tree** T_t for each resampling step t whose nodes are labeled with events, with the property that the children of a node v labeled with event A_v are labeled with events in $\Gamma^+(A_v) = \{A_v\} \cap \Gamma(A_v)$. The root of T_t is labeled with C_t; to construct the rest of the tree, we work backwards through $C_{t-1}, C_{t-2}, \ldots, C_1$, and for each event C_i we encounter we attach C_i as a child of the deepest v we can find with $C_i \in \Gamma^+(A_v)$, choosing arbitrarily if there is more than one such v, and discarding C_i if there is no such v.

Now we can ask what the probability is that we see some particular witness tree T in the execution log. Each vertex of T corresponds to some event A_v that we resample because it occurs; in order for it to occur, the previous assignments of each variable in $\mathrm{vbl}(A_v)$ must have made A_v true, which occurs with probability $\Pr[A_v]$. But since we resample all the variables in A_v, any subsequent assignments to these variables are independent of the ones that contributed to v; with sufficient handwaving (or a rather detailed coupling argument as found in [MT10]) this gives that each event A_v occurs with independent probability $\Pr[A_v]$, giving $\Pr[T] = \prod_{v \in T} \Pr[A_v]$.

Why do we care about this? Because every event we resample is the root of some witness tree, and we can argue that every event we resample is the root of a *distinct* witness tree. The proof is that since we only discard events B that have $\mathrm{vbl}(B)$ disjoint from all nodes already in the tree, once we put A at the root, any other instance of A gets included. So the witness tree rooted at the i-th occurrence of A in C will include exactly i copies of A, unlike the witness tree rooted at the j-th copy for $j \neq i$.

Now comes the sneaky trick: we'll count how many distinct witness trees T we can possibly have rooted at A, given that each occurs with probability $\prod_{v \in T} \Pr[A_v]$. This is done by constructing a **branching process** using the x_B values from Lemma 11.3.1 as probabilities of a node with label A having a kid labeled B for each B in $\Gamma^+(A)$, and doing algebraic manipulations on the resulting probabilities until $\prod_{v \in T} \Pr[A_v]$ shows up.

Formally, consider the process where we construct a tree by starting with a root labeled A, and for each vertex v with label A_v, giving it a child labeled B for each $B \in \Gamma^+(A_v)$ with independent probability x_B. We'll now calculate the probability p_T that this process generates a particular tree T.

Let $x'_B = x_B \prod_{C \in \Gamma(B)}(1 - x_C)$. Note that (11.3.1) says precisely that $\Pr[B] \leq x'_B$.

For each vertex v in T, let $W_v \subseteq \Gamma^+(A_v)$ be the set of events $B \in \Gamma^+(A_v)$ that *don't* occur as labels of children of v. The probability of getting T is equal to the product of the probabilities at each v of getting all of its children and none of its non-children. The non-children of v collectively contribute $\prod_{B \in W_v}(1 - x_B)$ to the product, and v itself contributes x_{A_v} (via the product for its parent), unless v is the root node. So we can express the giant product as

$$p_T = \frac{1}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in W_v} (1 - x_B) \right).$$

We don't like the W_v very much, so we get rid of them by pushing a $(1 - x_{A_v})$ up from each vertex v, which we compensate for by dividing by $1 - x_{A_v}$ at v itself, with an extra $1 - x_A$ showing up at the root. This gives

$$\begin{aligned} p_T &= \frac{1 - x_A}{x_A} \prod_{v \in T} \left(\frac{x_{A_v}}{1 - x_{A_v}} \prod_{B \in \Gamma^+(A_v)} (1 - x_B) \right). \\ &= \frac{1 - x_A}{x_A} \prod_{v \in T} \left(x_{A_v} \prod_{B \in \Gamma(A_v)} (1 - x_B) \right). \\ &= \frac{1 - x_A}{x_A} \prod_{v \in T} x'_{A_v}. \end{aligned}$$

Now we can bound the expected number of trees rooted at A that appear in C, assuming (11.3.1) holds. Letting $\mathcal{T}_A$ be the set of all such trees and

N_A the number that appear in C, we have

$$\begin{aligned}
\mathrm{E}\left[N_A\right] &= \sum_{T \in \mathcal{T}_A} \Pr\left[T \text{ appears in } C\right] \\
&\leq \sum_{T \in \mathcal{T}_A} \prod_{v \in T} \Pr\left[A(v)\right] \\
&\leq \sum_{T \in \mathcal{T}_A} \prod_{v \in T} x'_{A_v} \\
&= \sum_{T \in \mathcal{T}_A} \frac{x_A}{1 - x_A} p_T \\
&= \frac{x_A}{1 - x_A} \sum_{T \in \mathcal{T}_A} p_T \\
&\leq \frac{x_A}{1 - x_A}.
\end{aligned}$$

And we're done.

Chapter 12

Derandomization

Derandomization is the process of taking a randomized algorithm and turning it into a deterministic algorithm. This is useful both for practical reasons (deterministic algorithms are more predictable, which makes them easier to debug and gives hard guarantees on running time) and theoretical reasons (if we can derandomize any randomized algorithm we could show results like $\mathbf{P} = \mathbf{RP}$,[1] which would reduce the number of complexity classes that complexity theorists otherwise have to deal with). It may also be the case that derandomizing a randomized algorithm can be used for **probability amplification**, where we replace a low probability of success with a higher probability, in this case 1.

There are basically two approaches to derandomization:

1. Reduce the number of random bits used down to $O(\log n)$, and then search through all choices of random bits exhaustively. For example, if we only need pairwise independence, we could use the XOR technique from §5.1.2.1 to replace a large collection of variables with a small collection of random bits.

   Except for the exhaustive search part, this is how randomized algorithms are implemented in practice: rather than burning random bits continuously, a **pseudorandom generator** is initialized from a **seed** consisting of a small number of random bits. For pretty much all of the randomized algorithms we know about, we don't even need to use a particularly strong pseudorandom generator. This is largely because

[1] The class **RP** consists of all languages L for which there is a polynomial-time randomized algorithm that correctly outputs "yes" given an input x in L with probability at least 1/2, and never answers "yes" given an input x not in L. See §1.4.2 for a more extended description of **RP** and other randomized complexity classes.

current popular generators are the products of a process of evolution: pseudorandom generators that cause wonky behavior or fail to pass tests that approximate the assumptions made about them by typical randomized algorithms are abandoned in favor of better generators.[2]

From a theoretical perspective, pseudorandom generators offer the possibility of eliminating randomization from all randomized algorithms, except there is a complication. While (under reasonable cryptographic assumptions) there exist **cryptographically secure pseudorandom generators** whose output is indistinguishable from a genuinely random source by polynomial-time algorithms (including algorithms originally intended for other purposes), such generators are inherently incapable of reducing the number of random bits down to the $O(\log n)$ needed for exhaustive search. The reason is that any pseudorandom generator with only polynomially-many seeds can't be cryptographically secure, because we can distinguish it from a random source by just checking its output against the output for all possible seeds. Whether there is some other method for transforming an arbitrary algorithm in **RP** or **BPP** into a deterministic algorithm remains an open problem in complexity theory (and beyond the scope of this course).

2. Start with a specific randomized protocol and analyze its behavior enough that we can replace the random bits it uses with specific, deterministically-chosen bits we can compute. This is the main approach we will describe below. A non-constructive variant of this shows that we can always replace the random bits used by all inputs of a given size with a few carefully-selected fixed sequences (Adleman's Theorem, described in §12.2). More practical is the **method of conditional probabilities**, which chooses random bits sequentially based on which value is more likely to give a good outcome (see §12.4).

12.1 Deterministic vs. randomized algorithms

In thinking about derandomization, it can be helpful to have more than one way to look at a randomized algorithm. So far, we've describe randomized algorithms as random choices of deterministic algorithms ($M_r(x)$) or,

[2]Having cheaper computers helps here as well. Nobody would have been willing to spend 2496 bytes on the state vector for Mersenne Twister [MN98] back in 1975, but by 1998 this amount of memory was trivial for pretty much any computing device except the tiniest microcontrollers.

equivalently, as deterministic algorithms that happen to have random inputs ($M(r, x)$). This gives a very static view of how randomness is used in the algorithm. A more dynamic view is obtained by thinking of the computation of a randomized algorithm as a **computation tree**, where each path through the tree corresponds to a computation with a fixed set of random bits and a branch in the tree corresponds to a random decision. In either case we want an execution to give us the right answer with reasonably high probability, whether that probability measures our chance of getting a good deterministic machine for our particular input or landing on a good computation path.

12.2 Adleman's theorem

The idea of picking a good deterministic machine is the basis for **Adleman's theorem**[Adl78], a classic result in complexity theory. Adleman's theorem says that we can always replace randomness by an oracle that presents us with a fixed string of **advice** p_n that depends only on the size of the input n and has size polynomial in n. The formal statement of the theorem relates the class **RP**, which is the class of problems for which there exists a polynomial-time Turing machine $M(x, r)$ that outputs 1 at least half the time when $x \in L$ and never when $x \notin L$; and the class **P/poly**, which is the class of problems for which there is a polynomial-sized string p_n for each input size n and a polynomial-time Turing machine M' such that $M'(x, p_{|x|})$ outputs 1 if and only if $x \in L$.

Theorem 12.2.1. **RP** $\subseteq$ **P/poly**.

Proof. The intuition is that if any one random string has a constant probability of making M happy, then by choosing enough random strings we can make the probability that M fails using on every random string for any given input so small that even after we sum over all inputs of a particular size, the probability of failure is still small using the union bound (4.1.1). This is an example of **probability amplification**, where we repeat a randomized algorithm many times to reduce its failure probability.

Formally, consider any fixed input x of size n, and imagine running M repeatedly on this input with $n + 1$ independent sequences of random bits $r_1, r_2, \ldots, r_{n+1}$. If $x \notin L$, then $M(x, r_i)$ never outputs 1. If $x \in L$, then for each r_i, there is an independent probability of at least $1/2$ that $M(x, r_i) = 1$. So $\Pr[M(x, r_i) = 0] \leq 1/2$, and $\Pr[\forall i M(x, r_i) = 0] \leq 2^{-(n+1)}$. If we sum this probability of failure for each individual $x \in L$ of length n over the at

most 2^n such elements, we get a probability that any of them fail of at most $2^n 2^{-(n+1)} = 1/2$. Turning this upside down, any sequence of $n+1$ random inputs includes a **witness** that $x \in L$ for *all* inputs x with probability at least $1/2$. It follows that a good sequence $r_1, \ldots, r_{n+1}$, exists.

Our advice p_n is now some good sequence $p_n = \langle r_1 \ldots r_{n+1} \rangle$, and the deterministic advice-taking algorithm that uses it runs $M(x, r_i)$ for each r_i and returns true if and only if at least one of these executions returns true. □

The classic version of this theorem shows that anything you can do with a polynomial-size randomized circuit (a circuit made up of AND, OR, and NOT gates where some of the inputs are random bits, corresponding to the r input to M) can be done with a polynomial-size deterministic circuit (where now the p_n input is baked into the circuit, since we need a different circuit for each size n anyway).

A limitation of this result is that ordinary algorithms seem to be better described by **uniform** families of circuits, where there exists a polynomial-time algorithm that, given input n, outputs the circuit C_n for processing size-n inputs. In contrast, the class of circuits generated by Adleman's theorem is most likely **non-uniform**: the process of finding the good witnesses r_i is not something we know how to do in polynomial time (with the usual caveat that we can't prove much about what we can't do in polynomial time).

12.3 Limited independence

For some algorithms, it may be that full independence is not needed for all of the random bits. If the amount of independence needed is small enough, this may allow us to reduce the actual number of random bits consumed down to $O(\log n)$, at which point we can try all possible sequences of random bits in polynomial time.

Variants of this technique have been used heavily in the cryptography and complexity; see [LW05] for a survey of some early work in this area. We'll do a quick example of the method before before moving onto more direct approaches.

12.3.1 MAX CUT

Let's look at the randomized MAX CUT algorithm from §11.2.1. In the original algorithm, we use n independent random bits to assign the n vertices to S or T. The idea is that by assigning each vertex independently at

random to one side of the cut or the other, each edge appears in the cut with probability 1/2, giving a total of $m/2$ edges in the cut in expectation.

Suppose that we replace these n independent random bits with n pairwise-independent bits generated by taking XORs of subsets of $\lceil \lg(n+1) \rceil$ independent random bits as described in §5.1.2.1. Because the bits are pairwise-independent, the probability that the two endpoints of an edge are assigned to different sides of the cut is still exactly 1/2. So on average we get $m/2$ edges in the cut as before, and there is at least one sequence of random bits that guarantees a cut at least this big.

But with only $\lceil \lg(n+1) \rceil$ random bits, there are only $2^{\lceil \log(n+1) \rceil} < 2(n+1)$ possible sequences of random bits. If we try all of them, then we find a cut of size $m/2$ always. The total cost is $O(n(n+m))$ if we include the $O(n+m)$ cost of testing each cut. Note that this algorithm does not generate all 2^n possible cuts, but among those it *does* generate, there must be a large one.

In this particular case, we'll see below how to get the same result at a much lower cost, using more knowledge of the problem. So we have the typical trade-off between algorithm performance and algorithm designer effort.

12.4 The method of conditional probabilities

The **method of conditional probabilities** [Rag88] follows an execution of the randomized algorithm, but at each point where we would otherwise make a random decision, it makes a decision that minimizes the conditional probability of losing.

Structurally, this is similar to the method of bounded differences (see §5.3.3). Suppose our randomized algorithm generates m random values $X_1, X_2, \ldots, X_m$. Let $f(X_1, \ldots, X_m)$ be the indicator variable for our randomized algorithm failing (more generally, we can make it an expected cost or some other performance measure). Extend f to shorter sequences of values by defining $f(x_1, \ldots, x_k) = \mathrm{E}\left[f(x_1, \ldots, x_k, X_{k+1}, \ldots, X_m)\right]$. Then $Y_k = f(X_1, \ldots, X_k)$ is a Doob martingale, just as in the method of bounded differences. This implies that, for any partial sequence of values $x_1, \ldots, x_k$, there exists some next value x_{k+1} such that $f(x_1, \ldots, x_k) \geq f(x_1, \ldots, x_k, x_{k+1})$. If we can find this value, we can follow a path on which f always decreases, and obtain an outcome of the algorithm $f(x_1, \ldots, x_m)$ less than or equal to the initial value $f(\langle\rangle)$. If our outcomes are 0–1 (as in failure probabilities), and our initial value for f is less than 1, this means that we reach an outcome with $f = 0$.

The tricky part here is that it may be hard to compute $f(x_1, \ldots, x_k)$.

(It's always possible to do so in principle by enumerating all assignments of the remaining variables, but if we have time to do this, we can just search for a winning assignment directly.) What makes the method of conditional probabilities practical in many cases is that it's not necessary for f to compute the actual probability of failure, as long as (a) f gives an upper bound on the real probability of failure, at least in terminal configurations, and (b) f has the property used in the argument that for any partial sequence $x_1, \ldots, x_k$ there exists an extension $x_1, \ldots, x_k, x_{k+1}$ with $f(x_1, \ldots, x_k) \geq f(x_1, \ldots, x_k, x_{k+1})$. Such an f is called a **pessimistic estimator**. If we can find a pessimistic estimator that is easy to compute and starts out less than 1, then we can just follow it down the tree to a leaf that doesn't fail.

12.4.1 A trivial example

Here is a very bad randomized algorithm for generating a string of n zeros: flip n coins, and output the results. This has a $1-2^{-n}$ probability of failure, which is not very good.

We can derandomize this algorithm and get rid of all probability of failure at the same time. Let $f(x_1, \ldots, x_k)$ be the probability of failure after the first k bits. Then we can easily compute f (it's 1 if any of the bits are 1 and $1-2^{-(n-k)}$ otherwise). We can use this to find a bit x_{k+1} at each stage that reduces f. After noticing that all of these bits are zero, we improve the deterministic algorithm even further by leaving out the computation of f.

12.4.2 Deterministic construction of Ramsey graphs

Here is an example that is slightly less trivial. For this example we let f be a count of bad events rather than a failure probability, but the same method applies.

Recall from §11.1.2 that if $k \geq 3$, for $n \leq 2^{k/2}$ there exists a graph with n nodes and no cliques or independent sets of size k. The proof of this fact is to observe that each subset of k vertices is bad with probability 2^{-k+1}, and when $\binom{n}{k}2^{-k+1} < 1$, the expected number of bad subsets in $G_{n,1/2}$ is less than 1, showing that some good graph exists.

We can turn this into a deterministic $n^{O(\log n)}$ algorithm for finding a Ramsey graph in the worst case when $n = 2^{k/2}$. The trick is to set the edges to be present or absent one at a time, and for each edge, take the value that minimizes the expected number of bad subsets conditioned on the choices so far. We can easily calculate the conditional probability that

a subset is bad in $O(k)$ time: if it already has both a present and missing edge, it's not bad. Otherwise, if we've already set ℓ of its edges to the same value, it's bad with probability exactly $2^{-k+\ell}$. Summing this over all $O(n^k)$ subsets takes $O(n^k k)$ time per edge, and we have $O(n^2)$ edges, giving $O(n^{k+2}k) = O\left(n^{(2\lg n+2+\lg\lg n)}\right) = n^{O(\log n)}$ time total.

It's worth mentioning that there are better deterministic constructions in the literature. The best current construction that I am aware of (as of 2014) is given by an algorithm of Barak *et al.* [BRSW06], which constructs a graph with $k^{\log^c k}$ vertices with no clique or independent set of size k for any fixed c.

12.4.3 MAX CUT using conditional probabilities

Again we consider the algorithm from §11.2.1 that assigns vertices to S and T at random. To derandomize this algorithm, at each step we pick a vertex and assign it to the side of the cut that maximizes the conditional expectation of the number of edges that cross the cut. We can compute this conditional expectation as follows:

1. For any edge that already has both endpoints assigned to S or T, it's either in the cut or it isn't: add 0 or 1 as appropriate to the conditional expectation.

2. For any edge with only one endpoint assigned, there's a 1/2 probability that the other endpoint gets assigned to the other side (in the original randomized algorithm). Add 1/2 to the conditional expectation for these edges.

3. For any edge with neither endpoint assigned, we again have a 1/2 probability that it crosses the cut. Add 1/2 for these as well.

So now let us ask how assigning a particular previously unassigned vertex v to S or T affects the conditional probability. For any neighbor w of v that is not already assigned, adding v to S or T doesn't affect the 1/2 contribution of vw. So we can ignore these. The only effects we see are that if some neighbor w is in S, assigning v to S decreases the conditional expectation by 1/2 and assigning v to T increases the expectation by 1/2. So to maximize the conditional expectation, we should assign v to whichever side currently holds fewer of v's neighbors—the obvious greedy algorithm, which runs in $O(n+m)$ time if we are reasonably clever about keeping track of how many neighbors each unassigned node has in S and T. The advantage of the

method of conditional probabilities here is that we immediately get that the greedy algorithm achieves a cut of size $m/2$, which might require actual intelligence to prove otherwise.

12.4.4 Set balancing

Here we have a collection of vectors $v_1, v_2, \ldots, v_n$ in $\{0,1\}^m$. We'd like to find ± 1 coefficients $\epsilon_1, \epsilon_2, \ldots \epsilon_n$ that minimize $\max_j |X_j|$ where $X_j = \sum_{i=1}^n \epsilon_i v_{ij}$.

If we choose the ϵ_i randomly, Hoeffding's inequality (5.3.1) says for each fixed j that $\Pr[|X_j| > t] < 2\exp(-t^2/2n)$ (since there are at most n nonzero values v_{ij}). Setting $2\exp(-t^2/2n) \leq 1/m$ gives $t \geq \sqrt{2n \ln 2m}$. So by the union bound, we have $\Pr\left[\max_j |X_j| > \sqrt{2n \ln 2m}\right] < 1$: a solution exists.

Because there may be very complicated dependence between the X_j, it is difficult to calculate the probability of the event $\bigcup_j [|X_j| \geq t]$, whether conditioned on some of the ϵ_i or not. However, we can calculate the probability of the individual events $[|X_j| \geq t]$ exactly. Conditioning on $\epsilon_1, \ldots, \epsilon_k$, the expected value of X_j is just $\ell = \sum_{i=1}^k \epsilon_i v_{ij}$, and the distribution of $Y = X_j - \mathrm{E}[X_j]$ is the sum $r \leq n-k$ independent ± 1 random variables. So

$$\begin{aligned}\Pr[|X_j| > t \mid \epsilon_1, \ldots, \epsilon_k] &= 1 - \Pr[-t-\ell \leq Y \leq t-\ell] \\ &= \sum_{i=-t-\ell}^{t-\ell} \binom{r}{i} 2^{-r}.\end{aligned}$$

This last expression involves a linear number of terms, each of which we can calculate using a linear number of operations on integers that fit in a linear number of bits, so we can calculate the probability exactly in polynomial time by just adding them up.

For our pessimistic estimator, we take $U(\epsilon_1, \ldots, \epsilon_k) = \sum_{i=j}^n \Pr\left[|X_j| > \sqrt{2n \ln 2m} \;\middle|\; \epsilon_1, \ldots, \epsilon_k\right]$. Note that we can calculate each term in the sum by adding up a big pile of binomial coefficients. Furthermore, since each term in the sum is a Doob martingale, the sum is a martingale as well, so $\mathrm{E}[U(\epsilon_1, \ldots, \epsilon_{k+1}) \mid \epsilon_1, \ldots, \epsilon_k] = U(\epsilon_1, \ldots, \epsilon_k)$. It follows that for any choice of $\epsilon_1, \ldots, \epsilon_k$ there exists some ϵ_{k+1} such that $U(\epsilon_1, \ldots, \epsilon_k) \geq U(\epsilon_1, \ldots, \epsilon_{k+1})$, and we can determine this winning ϵ_{k+1} explicitly. Our previous argument shows that $U(\langle\rangle) < 1$, which implies that our final value $U(\epsilon_1, \ldots, \epsilon_n)$ will also be less than 1. Since U is integral, this means it must be 0, and we find an assignment in which $|X_j| < \sqrt{2n \ln 2m}$ for all j.

Chapter 13

Quantum computing

Quantum computing is a currently almost-entirely-theoretical branch of randomized algorithms that attempts to exploit the fact that probabilities at a microscopic scale arise in a mysterious way from more fundamental **probability amplitudes**, which are complex-valued and can in particular cancel each other out where probabilities can only add. In a quantum computation, we replace random bits with **quantum bits**—**qubits** for short—and replace random updates to the bits with quantum operations.

To explain how this works, we'll start by re-casting our usual model of a randomized computation to make it look more like the standard **quantum circuit** of Deutsch [Deu89]. We'll then get quantum computation by replacing all the real-valued probabilities with complex-valued amplitudes.

13.1 Random circuits

Let's consider a very simple randomized computer whose memory consists of two bits. We can describe our knowledge of the state of this machine using a vector of length 4, with the coordinates in the vector giving the probability of states 00, 01, 10, and 11. For example, if we know that the initial state is always 00, we would have the (column) vector

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Any such **state vector** x for the system must consist of non-negative real values that sum to 1; this is just the usual requirement for a discrete

probability space. Operations on the state consist of taking the old values of one or both bits and replacing them with new values, possibly involving both randomness and dependence on the old values. The law of total probability applies here, so we can calculate the new state vector x' from the old state vector x by the rule

$$x'_{b'_1 b'_2} = \sum_{b_1 b_2} x_{b_1 b_2} \Pr\left[X_{t+1} = b'_1 b'_2 \mid X_t = b_1 b_2\right].$$

These are linear functions of the previous state vector, so we can summarize the effect of our operation using a transition matrix A, where $x' = Ax$.[1]

For example, if we negate the second bit 2/3 of the time while leaving the first bit alone, we get the matrix

$$A = \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix}.$$

One way to derive this matrix other than computing each entry directly is that it is the **tensor product** of the matrices that represent the operations on each individual bit. The idea here is that the tensor product of A and B, written $A \otimes B$, is the matrix C with $C_{ij,k\ell} = A_{ik} B_{j\ell}$. We're cheating a little bit by allowing the C matrix to have indices consisting of pairs of indices, one for each of A and B; there are more formal definitions that justify this at the cost of being harder to understand.

In this particular case, we have

$$\begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1/3 & 2/3 \\ 2/3 & 1/3 \end{bmatrix}.$$

The first matrix in the tensor product gives the update rule for the first bit (the identity matrix—do nothing), while the second gives the update rule for the second.

[1] Note that this is the reverse of the convention we adopted for Markov chains in Chapter 9. There it was convenient to have $P_{ij} = p_{ij} = \Pr\left[X_{t+1} = j \mid X_t = i\right]$. Here we defer to the physicists and make the update operator come in front of its argument, like any other function.

Some operations are not decomposable in this way. If we swap the values of the two bits, we get the matrix

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which maps 00 and 11 to themselves but maps 01 to 10 and vice versa.

The requirement for all of these matrices is that they be **stochastic**. This means that each column has to sum to 1, or equivalently that $1A = 1$, where 1 is the all-ones vector. This just says that our operations map probability distributions to probability distributions; we don't apply an operation and find that the sum of our probabilities is now 3/2 or something. (Proof: If $1A = 1$, then $|Ax|_1 = 1(Ax) = (1A)x = 1x = |x|_1$.)

A **randomized computation** in this model now consists of a sequence of these stochastic updates to our random bits, and at the end performing a **measurement** by looking at what the values of the bits actually are. If we want to be mystical about it, we could claim that this measurement collapses the probability distribution over states representing our previous ignorance into a single unique state, but really we are opening the box to see what we got.

For example, we could generate two bias-2/3 coin-flips by starting with 00 and using the algorithm flip second, swap bits, flip second, or in matrix terms:

$$\begin{aligned} x_{\text{out}} &= ASAx_{\text{in}} \\ &= \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1/3 & 2/3 & 0 & 0 \\ 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 0 & 0 & 2/3 & 1/3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 1/9 \\ 2/9 \\ 2/9 \\ 4/9 \end{bmatrix}. \end{aligned}$$

When we look at the output, we find 11 with probability 4/9, as we'd expect, and similarly for the other values.

13.2 Bra-ket notation

A notational oddity that scares many people away from quantum mechanics in general and quantum computing in particular is the habit of practitioners in these fields of writing basis vectors and their duals using **bra-ket notation**, a kind of typographic pun invented by the physicist Paul Dirac [Dir39].

This is based on a traditional way of writing an **inner product** of two vectors x and y in "bracket form" as $\langle x|y\rangle$. The interpretation of this is $\langle x|y\rangle = x^*y$, where x^* is the **conjugate transpose** of x.

For real-valued x the conjugate transpose is the same as the transpose. For complex-valued x, each coordinate $x_i = a+bi$ is replaced by its **complex conjugate** $\bar{x}_i = a - bi$.) Using the conjugate transpose makes $\langle x|x\rangle$ equal $|x|_2^2$ when x is complex-valued.

For example, for our vector x_{in} above that puts all of its probability on 00, we have

$$\langle x_{\text{in}}|x_{\text{in}}\rangle = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = 1. \tag{13.2.1}$$

The typographic trick is to split in half both $\langle x|y\rangle$ and its expansion. For example, we could split (13.2.1) as

$$\langle x_{\text{in}}| = \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} \qquad\qquad |x_{\text{in}}\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}.$$

In general, wherever we used to have a bracket $\langle x|y\rangle$, we now have a **bra** $\langle x|$ and a **ket** $|y\rangle$. These are just row vectors and column vectors, with the relation that $\langle x|$ is always the conjugate transpose of $|x\rangle$.

The second trick in bra-ket notation is to make the contents of the bra or ket an arbitrary name. For kets, this will usually describe some state. As an example, we might write x_{in} as $|00\rangle$ to indicate that it's the basis vector that puts all of its weight on the state 00. For bras, this is the linear operator that returns 1 when applied to the given state and 0 when applied to any orthogonal state. So $\langle 00|\,|00\rangle = \langle 00|00\rangle = 1$ but $\langle 00|\,|01\rangle = \langle 00|01\rangle = 0$.

13.2.1 States as kets

This notation is useful for the same reason that variable names are useful. It's a lot easier to remember that $|01\rangle$ refers to the distribution assigning

probability 1 to state 01 than that $\begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix}^\top$ is.

Non-basis vectors we can express using a linear combination of kets. For example, we can write

$$x_{\text{out}} = \frac{1}{9}\left|00\right\rangle + \frac{2}{9}\left|01\right\rangle + \frac{2}{9}\left|10\right\rangle + \frac{4}{9}\left|11\right\rangle.$$

This is not as compact as the vector, but has the advantage of clearly labeling what states the probabilities apply to.

13.2.2 Operators as sums of kets times bras

A similar trick can be used to express operators, like the swap operator S. We can represent S as a combination of maps from specific states to specific other states. For example, the operator

$$\left|01\right\rangle\left\langle 10\right| = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 0\\0 & 0 & 0 & 0\end{bmatrix}$$

maps $\left|10\right\rangle$ to $\left|01\right\rangle$ (Proof: $\left|01\right\rangle\left\langle 10\right|\left|10\right\rangle = \left|01\right\rangle\left\langle 10|10\right\rangle = \left|01\right\rangle$) and sends all other states to 0. Add up four of these mappings to get

$$S = \left|00\right\rangle\left\langle 00\right| + \left|10\right\rangle\left\langle 01\right| + \left|01\right\rangle\left\langle 10\right| + \left|11\right\rangle\left\langle 11\right| = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix}.$$

Here the bra-ket notation both labels what we are doing and saves writing a lot of zeros.

13.3 Quantum circuits

So how do we turn our random circuits into quantum circuits?

The first step is to replace our random bits with quantum bits (qubits).

For a single random bit, the state vector represents a probability distribution

$$p_0\left|0\right\rangle + p_1\left|1\right\rangle,$$

where p_0 and p_1 are non-negative real numbers with $p_0 + p_1 = 1$.

For a single qubit, the state vector represents amplitudes

$$a_0 \left|0\right\rangle + a_1 \left|1\right\rangle,$$

where a_0 and a_1 are complex numbers with $\left|a_0\right|^2 + \left|a_1\right|^2 = 1$.[2] The reason for this restriction on amplitudes is that if we measure a qubit, we will see state 0 with probability $\left|a_0\right|^2$ and state 1 with probability $\left|a_1\right|^2$. Unlike with random bits, these probabilities are not mere expressions of our ignorance but arise through a still somewhat mysterious process from the more fundamental amplitudes.[3]

With multiple bits, we get amplitudes for all combinations of the bits, e.g.

$$\frac{1}{2}\left(\left|00\right\rangle + \left|01\right\rangle + \left|10\right\rangle + \left|11\right\rangle\right)$$

gives a state vector in which each possible measurement will be observed with equal probability $\left(\frac{1}{2}\right)^2 = \frac{1}{4}$. We could also write this state vector as

$$\begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}.$$

[2]The **absolute value**, **norm**, or **magnitude** $|a + bi|$ of a complex number is given by $\sqrt{a^2 + b^2}$. When $b = 0$, this is the same as the absolute value for the corresponding real number. For any complex number x, the norm can also be written as $\sqrt{\bar{x}x}$, where $\bar{x}$ is the complex conjugate of x. This is because $\sqrt{(a + bi)(a - bi)} = \sqrt{a^2 - (bi)^2} = \sqrt{a^2 + b^2}$. The appearance of the complex conjugate here explains why we define $\langle x|y\rangle = x^* y$; the conjugate transpose means that for $\langle x|x\rangle$, when we multiply x_i^* by x_i we are computing a squared norm.

[3]In the old days of "shut up and calculate," this process involved the unexplained power of an observer to collapse a superposition into a classical state. Nowadays the most favored explanation involves **decoherence**, the difficulty of maintaining superpositions in systems that are interacting with large, warm objects with lots of thermodynamic degrees of freedom (measuring instruments, brains). The decoherence explanation is particularly useful for explaining why real-world quantum computers have a hard time keeping their qubits mixed even when nobody is looking at them. Decoherence by itself does not explain *which* basis states a system collapses to. Since bases in linear algebra are pretty much arbitrary, it would seem that we could end up running into a physics version of Goodman's grue-bleen paradox [Goo83], but there are ways of dealing with this too (**einselection**). Since all of this is (a) well beyond my own limited comprehension of quantum mechanics and (b) irrelevant to the theoretical model we are using, these issues will not be discussed further.

13.3.1 Quantum operations

In the random circuit model, at each step we pick a small number of random bits, apply a stochastic transformation to them, and replace their values with the results. In the quantum circuit model, we do the same thing, but now our transformations must have the property of being **unitary**. Just as a stochastic matrix preserves the property that the probabilities in a state vector sum to 1, a **unitary matrix** preserves the property that the squared norms of the amplitudes in a state vector sum to 1.

Formally, a square, complex matrix A is unitary if it preserves inner products: $\langle Ax|Ay\rangle = \langle x|y\rangle$ for all x and y. Alternatively, A is unitary if $A^*A = AA^* = I$[4], because $\langle Ax|Ay\rangle = (Ax)^*(Ay) = x^*A^*Ay = x^*Iy = x^*y = \langle x|y\rangle$. Yet another way to state this is that the columns of A from an orthonormal basis: this means that $\langle A_i|A_j\rangle = 0$ if $i \neq j$ and 1 if $i = j$, which is just a more verbose way to say $A^*A = I$. The same thing also works if we consider rows instead of columns.

The rule then is: at each step, we can operate on some constant number of qubits by applying a unitary transformation to them. In principle, this could be *any* unitary transformation, but some particular transformations show up often in actual quantum algorithms.[5]

The simplest unitary transformations are permutations on states (like the operator that swaps two qubits), and rotations of a single state. One particularly important rotation is the **Hadamard operator**

$$H = \sqrt{\frac{1}{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

This maps $|0\rangle$ to the superposition $\sqrt{\frac{1}{2}}\,|0\rangle + \sqrt{\frac{1}{2}}\,|1\rangle$; since this superposition collapses to either $|0\rangle$ or $|1\rangle$ with probability 1/2, the state resulting from $H\,|0\rangle$ is the quantum-computing equivalent of a fair coin-flip. Note that $H\,|1\rangle = \sqrt{\frac{1}{2}}\,|0\rangle - \sqrt{\frac{1}{2}}\,|1\rangle \neq H\,|0\rangle$, even though both yield the same probabilities; these two superpositions have different **phases** and may behave differently when operated on further. This is necessary: all quantum operations are **reversible**, because any unitary matrix U has an inverse U^*.

If we apply H in parallel to all the qubits in our system, we get the n-fold tensor product $H^{\otimes n}$, which (if we take our bit-vector indices as integers

[4] Recall that A^* is the conjugate transpose of A.

[5] Deutsch's original paper [Deu89] shows that repeated applications of single-qubit rotations and the CNOT operation (described in §13.3.2) are enough to approximate any unitary transformation.

$0 \ldots N-1 = 2^n - 1$ represented in binary) maps $|0\rangle$ to $\sqrt{\frac{1}{N}} \sum_{i=0}^{N-1} |i\rangle$. So n applications of H effectively scramble a deterministic initial state into a uniform distribution across all states. We'll see this scrambling operation again when we look at Grover's algorithm in §13.5.

13.3.2 Quantum implementations of classical operations

One issue that comes up with trying to implement classical algorithms in the quantum-circuit model is that classical operation are generally not reversible: if I execution $x \leftarrow x \wedge y$, it may not be possible to reconstruct the old state of x. So I can't implement AND directly as a quantum operation.

The solution is to use more sophisticated reversible operations from which standard classical operations can be extracted as a special case. A simple example is the **controlled NOT** or **CNOT** operator, which computes the mapping $(x, y) \mapsto (x, x \oplus y)$. This corresponds to the matrix (over the basis $|00\rangle, |01\rangle, |10\rangle, |11\rangle$)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

which is clearly unitary (the rows are just the standard basis vectors). We could also write this more compactly as $|00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11|$.

The CNOT operator gives us XOR, but for more destructive operations we need to use more qubits, possibly including junk qubits that we won't look at again but that are necessary to preserve reversibility. The **Toffoli gate** or **controlled controlled NOT** gate (**CCNOT**) is a 3-qubit gate that was originally designed to show that classical computation could be performed reversibly [Tof80]. It implements the mapping $(x, y, z) \mapsto (x, y, (x \wedge y) \oplus z)$, which corresponds to the 8×8 matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

By throwing in some extra qubits we don't care about, Toffoli gates can implement basic operations like NAND ($(x, y, 1) \mapsto (x, y, \neg(x \wedge y))$), NOT ($(x, 1, 1) \mapsto (x, 1, \neg x)$), and fan-out ($(x, 1, 0) \mapsto (x, 1, x)$).[6] This gives a sufficient basis for implementing all classical circuits.

13.3.3 Representing Boolean functions

Suppose we have a quantum circuit that computes a Boolean function f. There are two conventions for representing the output of f:

1. We can represent $f(x)$ by XORing it with an extra qubit y: $|x, y\rangle \mapsto |x, y \oplus f(x)\rangle$. This is the approach taken by the CNOT ($f(x) = x$) and CCNOT ($f(x_1 x_2) = x_1 \wedge x_2$) gates.

2. We can represent $f(x)$ by changing the phase of $|x\rangle$: $|x\rangle \mapsto (-1)^{f(x)} |x\rangle$. The actual unitary operator corresponding to this is $\sum_x (-1)^{f(x)} |x\rangle \langle x|$, which in matrix form looks like a truth table for f expressed as ± 1 values along the diagonal, e.g.:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   for the XOR function.

   This has the advantage of requiring one fewer qubit, but we can't observe the value of $f(x)$ directly, because the amplitude of $|x\rangle$ is unchanged. Where this comes in handy is when we can use the change in phase in the context of a larger quantum algorithm to get cancellations for some values of x.

The first representation makes more sense for modeling classical circuits. The second turns out to be more useful when f is a subroutine in a larger

[6] In the case of fan-out, this only works with perfect accuracy for classical bits and not superpositions, which run into something called the **no-cloning theorem**. For example, applying CCNOT to $\frac{1}{\sqrt{2}} |010\rangle + \frac{1}{\sqrt{2}} |110\rangle$ yields $\frac{1}{\sqrt{2}} |010\rangle + \frac{1}{\sqrt{2}} |111\rangle$. This works, sort of, but the problem is that the first and last bits are still entangled, meaning we can't operate on them independently. This is actually not all that different from what happens in the probabilistic case (if I make a copy of a random variable X, it's correlated with the original X), but it has good or bad consequences depending on whether you want to prevent people from stealing your information undetected or run two copies of a quantum circuit on independent replicas of a single superposition.

quantum algorithm. Fortunately, the operator with matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

converts the $|0\rangle$–$|1\rangle$ representation into the ± 1 representation, so we can build any desired classical f using 0–1 logic and convert it to ± 1 as needed.

13.3.4 Practical issues (which we will ignore)

The big practical question is whether any of these operations—or even non-trivial numbers of independently manipulable qubits—can be implemented in a real, physical system. As theorists, we can ignore these issues, but in real life they are what would make quantum computing science instead of science fiction.[7]

13.3.5 Quantum computations

Putting everything together, a quantum computation consists of three stages:

1. We start with a collection of qubits in some known state x_0 (e.g., $|000\ldots0\rangle$).
2. We apply a sequence of unitary operators $A_1, A_2, \ldots A_m$ to our qubits.
3. We take a measurement of the final superposition $A_m A_{m-1} \ldots A_1 x_0$ that collapses it into a single state, with probability equal to the square of the amplitude of that state.

Our goal is for this final state to tell us what we want to know, with reasonably high probability.

13.4 Deutsch's algorithm

We now have enough machinery to describe a real quantum algorithm. Known as Deutsch's algorithm, this computes $f(0) \oplus f(1)$ while evaluating f once [Deu89]. The trick, of course, is that f is applied to a superposition.

[7] Currently known results, sadly, still put quantum computing mostly in the science fiction category.

Assumption: f is implemented reversibly, as a quantum computation that maps $|x\rangle$ to $(-1)^{f(x)}|x\rangle$. To compute $f(0) \oplus f(1)$, evaluate

$$\begin{aligned} HfH|0\rangle &= \sqrt{\frac{1}{2}} Hf\left(|0\rangle + |1\rangle\right) \\ &= \sqrt{\frac{1}{2}} H\left((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right) \\ &= \frac{1}{2}\left(\left((-1)^{f(0)} + (-1)^{f(1)}\right)|0\rangle + \left((-1)^{f(0)} - (-1)^{f(1)}\right)|1\rangle\right). \end{aligned}$$

Suppose now that $f(0) = f(1) = b$. Then the $|1\rangle$ terms cancel out and we are left with

$$\frac{1}{2}\left(2 \cdot (-1)^b |0\rangle\right) = (-1)^b |0\rangle.$$

This puts all the weight on $|0\rangle$, so when we take our measurement at the end, we'll see 0.

Alternatively, if $f(0) = b \neq f(1)$, it's the $|0\rangle$ terms that cancel out, leaving $(-1)^b|1\rangle$. Again the phase depends on b, but we don't care about the phase: the important thing is that if we measure the qubit, we always see 1.

The result in either case is that with probability 1, we determine the value of $f(0) \oplus f(1)$, after evaluating f once (albeit on a superposition of quantum states).

This is kind of a silly example, because the huge costs involved in building our quantum computer probably swamp the factor-of-2 improvement we got in the number of calls to f. But a generalization of this trick, known as the Deutsch-Josza algorithm [DJ92], solves the much harder (although still a bit contrived-looking) problem of distinguishing a constant Boolean function on n bits from a function that outputs one for exactly half of its inputs. No deterministic algorithm can solve this problem without computing at least $2^n/2 + 1$ values of f, giving an exponential speed-up.[8]

[8] The speed-up compared to a randomized algorithm that works with probability $1 - \epsilon$ is less impressive. With randomization, we only need to look at $O(\log 1/\epsilon)$ values of f to see both a 0 and a 1 in the non-constant case. But even here, the Deutsch-Josza algorithm does have the advantage of giving the correct answer always.

13.5 Grover's algorithm

Grover's algorithm [Gro96] is one of two main exemplars for the astonishing power of quantum computers.[9] The idea of Grover's algorithm is that if we have a function f on $N = 2^n$ possible inputs whose value is 1 for exactly one possible input w, we can find this w with high probability using $O(\sqrt{N})$ quantum evaluations of f. As with Deutsch's algorithm, we assume that f is encoded as an operator (conventionally written U_w) that maps each $|x\rangle$ to $(-1)^{f(x)}|x\rangle$.

The basic outline of the algorithm:

1. Start in the superposition $|s\rangle = \sqrt{\frac{1}{N}}\sum_x |x\rangle = H^{\otimes n}|0\rangle$.
2. Alternate between applying the **Grover diffusion operator** $D = 2|s\rangle\langle s| - I$ and the f operator $U_w = 2|w\rangle\langle w| - I$. Do this $O(\sqrt{n})$ times (the exact number of iterations is important and will be calculated below).
3. Take a measurement of the state. It will be w with high probability.

The details involving showing (a) that we can generate the original superposition $|s\rangle$, (b) that we can implement D efficiently using unitary operations on a constant number of qubits each, and (c) that we actually get w at the end of this process.

13.5.1 Initial superposition

To get the initial superposition, start with $|0^n\rangle$ and apply the Hadamard transform to each bit individually; this gives $\sqrt{\frac{1}{N}}\sum_x |x\rangle$ as claimed.

13.5.2 The Grover diffusion operator

We have the definition $D = 2|s\rangle\langle s| - I$.

Before we try to implement this, let's start by checking that it is in fact

[9] The other is Shor's algorithm [Sho97], which allows a quantum computer to factor n-bit integers in time polynomial in n. Sadly, Shor's algorithm is a bit too complicated to talk about here.

unitary. Compute

$$\begin{aligned} DD^* &= (2\left|s\right\rangle\left\langle s\right| - I)^2 \\ &= 4\left|s\right\rangle\left\langle s\right|\left|s\right\rangle\left\langle s\right| - 4\left|s\right\rangle\left\langle s\right| + I^2 \\ &= 4\left|s\right\rangle\left\langle s\right| - 4\left|s\right\rangle\left\langle s\right| + I \\ &= I. \end{aligned}$$

Here we use the fact that $\left|s\right\rangle\left\langle s\right|\left|s\right\rangle\left\langle s\right| = \left|s\right\rangle\left\langle s|s\right\rangle\left\langle s\right| = \left|s\right\rangle(1)\left\langle s\right| = \left|s\right\rangle\left\langle s\right|$.

Recall that $\left|s\right\rangle = H^{\otimes n}\left|0^n\right\rangle$, where $H^{\otimes n}$ is the result of applying H to each of the n bits individually. We also have that $H^* = H$ and $HH^* = I$, from which $H^{\otimes n}H^{\otimes n} = I$ as well.

So we can expand

$$\begin{aligned} D &= 2\left|s\right\rangle\left\langle s\right| - I \\ &= 2H^{\otimes n}\left|0^n\right\rangle(H^{\otimes n}\left|0^n\right\rangle)^* - I \\ &= 2H^{\otimes n}\left|0^n\right\rangle\left\langle 0^n\right|H^{\otimes n} - I \\ &= H^{\otimes n}(2\left|0^n\right\rangle\left\langle 0^n\right| - I)H^{\otimes n}. \end{aligned}$$

The two copies of $H^{\otimes n}$ involve applying H to each of the n bits, which we can do. The operator in the middle, $2\left|0^n\right\rangle\left\langle 0^n\right| - I$, maps $\left|0^n\right\rangle$ to $\left|0^n\right\rangle$ and maps all other basis vectors $\left|x\right\rangle$ to $-\left|x\right\rangle$. This can be implemented as a NOR of all the qubits, which we can do using our tools for classical computations. So the entire operator D can be implemented using $O(n)$ qubit operations, most of which can be done in parallel.

13.5.3 Effect of the iteration

To see what happens when we apply U_wD, it helps to represent the state in terms of a particular two-dimensional basis. The idea here is that the initial state $\left|s\right\rangle$ and the operation U_wD are symmetric with respect to any basis vectors $\left|x\right\rangle$ that aren't $\left|w\right\rangle$, so instead of tracking all of these non-w vectors separately, we will represent all of them by a single composite vector

$$\left|u\right\rangle = \sqrt{\frac{1}{N-1}}\sum_{x\neq w}\left|x\right\rangle.$$

The coefficient $\sqrt{\frac{1}{N-1}}$ is chosen to make $\langle u|u\rangle = 1$. As always, we like our vectors to have length 1.

Using $|u\rangle$, we can represent

$$|s\rangle = \sqrt{\frac{1}{N}}\,|w\rangle + \sqrt{\frac{N-1}{N}}\,|u\rangle\,. \tag{13.5.1}$$

A straightforward calculation shows that this indeed puts $\sqrt{\frac{1}{N}}$ amplitude on each $|x\rangle$.

Now we're going to bring in some trigonometry. Let $\theta = \sin^{-1}\sqrt{\frac{1}{N}}$, so that $\sin\theta = \sqrt{\frac{1}{N}}$ and $\cos\theta = \sqrt{1-\sin^2\theta} = \sqrt{\frac{N-1}{N}}$. We can then rewrite (13.5.1) as

$$|s\rangle = (\sin\theta)\,|w\rangle + (\cos\theta)\,|u\rangle\,. \tag{13.5.2}$$

Let's look at what happens if we expand D using (13.5.2):

$$\begin{aligned}
D &= 2\,|s\rangle\langle s| - I \\
&= 2\left((\sin\theta)\,|w\rangle + (\cos\theta)\,|u\rangle\right)\left((\sin\theta)\,\langle w| + (\cos\theta)\,\langle u|\right) - I \\
&= (2\sin^2\theta - 1)\,|w\rangle\langle w| + (2\sin\theta\cos\theta)\,|w\rangle\langle u| + (2\sin\theta\cos\theta)\,|u\rangle\langle w| + (2\cos^2\theta - 1)\,|u\rangle\langle u| \\
&= (-\cos 2\theta)\,|w\rangle\langle w| + (\sin 2\theta)\,|w\rangle\langle u| + (\sin 2\theta)\,|u\rangle\langle w| + (\cos 2\theta)\,|u\rangle\langle u| \\
&= \begin{bmatrix} -\cos 2\theta & \sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix},
\end{aligned}$$

where the matrix is over the basis $(|w\rangle, |u\rangle)$.

Multiplying by U_w negates all the $|w\rangle$ coefficients. So we get

$$\begin{aligned}
U_w D &= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} -\cos 2\theta & \sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix} \\
&= \begin{bmatrix} \cos 2\theta & -\sin 2\theta \\ \sin 2\theta & \cos 2\theta \end{bmatrix}.
\end{aligned} \tag{13.5.3}$$

Aficionados of computer graphics, robotics, or just matrix algebra in general may recognize (13.5.3) as the matrix that rotates two-dimensional vectors by 2θ. Since we started with $|s\rangle$ at an angle of θ, after t applications of this matrix we will be at an angle of $(2t+1)\theta$, or in state

$$(\sin(2t+1)\theta)\,|w\rangle + (\cos(2t+1)\theta)\,|u\rangle\,.$$

Ideally, we pick t so that $(2t+1)\theta = \pi/2$, which would put all of the amplitude on $|w\rangle$. Because t is an integer, we can't do this exactly, but

setting $t = \left\lfloor \frac{\pi/2\theta - 1}{2} \right\rfloor$ will get us somewhere between $\pi/2 - 2\theta$ and $\pi/2$. Since $\theta \approx \sqrt{\frac{1}{N}}$, this gives us a probability of seeing $|w\rangle$ in our final measurement of $1 - O(\sqrt{1/N})$ after $O(\sqrt{N})$ iterations of $U_w D$.

Sadly, this is as good as it gets. A lower bound of Bennet *et al.* [BBBV97] shows that *any* quantum algorithm using U_w as the representation for f must apply U_w $\Omega(\sqrt{N})$ times to find w. So we get a quadratic speedup but not the exponential speedup we'd need to solve **NP**-complete problems directly.

Appendix A

Assignments

Assignments are typically due Wednesdays at 17:00. Assignments should be submitted in PDF format via the classesv2 Drop Box.

A.1 Assignment 1: due Wednesday, 2014-09-10, at 17:00

A.1.1 Bureaucratic part

Send me email! My address is `james.aspnes@gmail.com`.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

A.1.2 Two terrible data structures

Consider the following data structures for maintaining a sorted list:

1. A sorted array A. Inserting a new element at position $A[i]$ into an array that current contains k elements requires moving the previous

values in $A[i] \ldots A[k]$ to $A[i+1] \ldots A[k+1]$. Each element moved costs one unit.

For example, if the array currently contains `1 3 5`, and we insert `2`, the resulting array will contain `1 2 3 5` and the cost of the operation will be 2, because we had to move `3` and `5`.

2. A sorted doubly-linked list. Inserting a new element at a new position requires moving the head pointer from its previous position to a neighbor of the new node, and then to the new node; each pointer move costs one unit.

   For example, if the linked list currently contains `1 3 5`, with the head pointer pointing to `5`, and we insert `2`, the resulting linked list will contain `1 2 3 5`, with the head pointing to `2`, and the cost will be 2, because we had to move the pointer from `5` to `3` and then from `3` to `2`. Note that we do not charge for updating the pointers between the new element and its neighbors. We will also assume that inserting the first element is free.

Suppose that we insert the elements 1 through n, in random order, into both data structures. Give an exact closed-form expression for the expected cost of each sequence of insertions.

Solution

1. Suppose we have already inserted k elements. Then the next element is equally likely to land in any of the positions $A[1]$ through $A[k+1]$. The number of displaced elements is then uniformly distributed in 0 through k, giving an expected cost for this insertion of $\frac{k}{2}$.

   Summing over all insertions gives

$$\begin{aligned}\sum_{k=0}^{n-1} \frac{k}{2} &= \frac{1}{2} \sum_{k=0}^{n-1} \\ &= \frac{n(n-1)}{4}.\end{aligned}$$

   An alternative proof, which also uses linearity of expectation, is to define X_{ij} as the indicator variable for the event that element j moves when element $i < j$ is inserted. This is 1 if an only if j is inserted before i, which by symmetry occurs with probability exactly 1/2. So

the expected total number of moves is

$$\begin{aligned}\sum_{1\le i<j\le n} \mathrm{E}\left[X_{ij}\right] &= \sum_{1\le i<j\le n} \frac{1}{2} \\ &= \frac{1}{2}\binom{n}{2} \\ &= \frac{n(n-1)}{4}.\end{aligned}$$

It's always nice when we get the same answer in situations like this.

2. Now we need to count how far the pointer moves between any two consecutive elements. Suppose that we have already inserted $k-1>0$ elements, and let X_k be the cost of inserting the k-th element. Let i and j be the indexes in the sorted list of the new and old pointer positions after the k-th insertion. By symmetry, all pairs of distinct

positions $i \neq j$ are equally likely. So we have

$$\begin{aligned}
\mathrm{E}\left[X_k\right] &= \frac{1}{k(k-1)} \sum_{i \neq j} |i-j| \\
&= \frac{1}{k(k-1)} \left(\sum_{1 \leq i < j \leq k} (j-i) + \sum_{1 \leq j < i \leq k} (i-j) \right) \\
&= \frac{2}{k(k-1)} \sum_{1 \leq i < j \leq k} (j-i) \\
&= \frac{2}{k(k-1)} \sum_{j=1}^{k} \sum_{\ell=1}^{j-1} \ell \\
&= \frac{2}{k(k-1)} \sum_{j=1}^{k} \frac{j(j-1)}{2} \\
&= \frac{1}{k(k-1)} \sum_{j=1}^{k} j(j-1). \\
&= \frac{1}{k(k-1)} \sum_{j=1}^{k} \left(j^2 - j\right) \\
&= \frac{1}{k(k-1)} \cdot \left(\frac{(2k+1)k(k+1)}{6} - \frac{k}{k+1} 2 \right) \\
&= \frac{1}{k(k-1)} \cdot \frac{(2k-2)k(k+1)}{6} \\
&= \frac{k+1}{3}.
\end{aligned}$$

This is such a simple result that we might reasonably expect that there is a faster way to get it, and we'd be right. A standard trick is to observe that we can simulate choosing k points uniformly at random from a line of n points by instead choosing $k+1$ points uniformly at random from a cycle of $n+1$ points, and deleting the first point chosen to turn the cycle back into a line. In the cycle, symmetry implies that the expected distance between each point and its successor is the same as for any other point; there are $k+1$ such distances, and they add up to $n+1$, so each expected distance is exactly $\frac{n+1}{k+1}$.

In our particular case, n (in the formula) is k and k (in the formula) is 2, so we get $\frac{k+1}{3}$. Note we are sweeping the whole absolute value

thing under the carpet here, so maybe the more explicit derivation is safer.

However we arrive at $\mathrm{E}[X_k] = \frac{k+1}{3}$ (for $k > 1$), we can sum these expectations to get our total expected cost:

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=2}^{n} X_k\right] &= \sum_{k=2}^{n} \mathrm{E}[X_k] \\
&= \sum_{k=2}^{n} \frac{k+1}{3} \\
&= \frac{1}{3}\sum_{\ell=3}^{n+1} \ell \\
&= \frac{1}{3}\left(\frac{(n+1)(n+2)}{2} - 3\right) \\
&= \frac{(n+1)(n+2)}{6} - 1.
\end{aligned}$$

It's probably worth checking a few small cases to see that this answer actually makes sense.

For large n, this shows that the doubly-linked list wins, but not by much: we get roughly $n^2/6$ instead of $n^2/4$. This is a small enough difference that in practice it is probably dominated by other constant-factor differences that we have neglected.

A.1.3 Parallel graph coloring

Consider the following algorithm for assigning one of k colors to each node in a graph with m edges:

1. Assign each vertex u a color c_u, chosen uniformly at random from all k possible colors.

2. For each vertex u, if u has a neighbor v with $c_u = c_v$, assign u a new color c'_u, again chosen uniformly at random from all k possible colors. Otherwise, let $c'_u = c_u$.

Note that any new colors c' do not affect the test $c_u = c_v$. A node changes its color only if it has the same original color as the original color of one or more of its neighbors.

Suppose that we run this algorithm on an r-regular[1] triangle-free[2] graph. As a function of k, r, and m, give an exact closed-form expression for the expected number of monochromatic[3] edges after running both steps of the algorithm.

Solution

We can use linearity of expectation to compute the probability that any particular edge is monochromatic, and then multiply by m to get the total.

Fix some edge uv. If either of u or v is recolored in step 2, then the probability that $c'_u = c'_v$ is exactly $1/k$. If neither is recolored, the probability that $c'_u = c'_v$ is zero (otherwise $c_u = c_v$, forcing both to be recolored). So we can calculate the probability that $c'_u = c'_v$ by conditioning on the event A that neither vertex is recolored.

This event occurs if both u and v have no neighbors with the same color. The probability that $c_u = c_v$ is $1/k$. The probability that any particular neighbor w of u has $c_w = c_u$ is also $1/k$; similarly for any neighbor w of v. These events are all independent on the assumption that the graph is triangle-free (which implies that no neighbor of u is also a neighbor of v). So the probability that none of these $2r-1$ events occur is $(1-1/k)^{2r-1}$.

We then have

$$\begin{aligned}\Pr[c_u = c_v] &= \Pr\left[c_u = c_v \;\middle|\; \overline{A}\right]\Pr\left[\overline{A}\right] + \Pr[c_u = c_v \mid A]\Pr[A] \\ &= \frac{1}{k}\cdot\left(1-\left(1-\frac{1}{k}\right)^{2r-1}\right).\end{aligned}$$

Multiply by m to get

$$\frac{m}{k}\cdot\left(1-\left(1-\frac{1}{k}\right)^{2r-1}\right).$$

For large k, this is approximately $\frac{m}{k}\cdot\left(1-e^{-(2r-1)/k}\right)$, which is a little bit better than than the $\frac{m}{k}$ expected monochrome edges from just running step 1.

Repeated application of step 2 may give better results, particular if k is large relative to r. We will see this technique applied to a more general class of problems in §11.3.5.

[1] Every vertex has exactly r neighbors.

[2] There are no vertices u, v, and w such that all are neighbors of each other.

[3] Both endpoints have the same color.

A.2 Assignment 2: due Wednesday, 2014-09-24, at 17:00

A.2.1 Load balancing

Suppose we distribute n processes independently and uniformly at random among m machines, and pay a communication cost of 1 for each pair of processes assigned to different machines. Let C be the total communication cost, that is, the number of pairs of processes assigned to different machines. What is the expectation and variance of C?

Let C_{ij} be the communication cost between machines i and j. This is just an indicator variable for the event that i and j are assigned to different machines, which occurs with probability $1-\frac{1}{m}$. We have $C = \sum_{1\le i<j\le n} C_{ij}$.

1. Expectation is a straightforward application of linearity of expectation. There are $\binom{n}{2}$ pairs of processes, and $\mathrm{E}[C_{ij}] = 1 - \frac{1}{m}$ for each pair, so

$$\mathrm{E}[C] = \binom{n}{2}\left(1 - \frac{1}{m}\right).$$

2. Variance is a little trickier because the C_{ij} are not independent. But they are pairwise independent: even if we fix the location of i and j, the expectation of C_{jk} is still $1 - \frac{1}{m}$, so $\mathrm{Cov}[C_{ij}, C_{jk}] = \mathrm{E}[C_{ij}C_{jk}] - \mathrm{E}[C_{ij}] \cdot \mathrm{E}[C_{jk}] = 0$. So we can compute

$$\mathrm{Var}[C] = \sum_{1\le i<j<n} \mathrm{Var}[C_{ij}] = \binom{n}{2}\frac{1}{m}\left(1 - \frac{1}{m}\right).$$

A.2.2 A missing hash function

A clever programmer inserts X_i elements in each of m buckets in a hash table, where each bucket i is implemented as a balanced binary search tree with search cost at most $\lceil \lg(X_i + 1)\rceil$. We are interested in finding a particular target element x, which is equally likely to be in any of the buckets, but we don't know what the hash function is.

Suppose that we know that the X_i are independent and identically distributed with $\mathrm{E}[X_i] = k$, and that the location of x is independent of the values of the X_i. What is best upper bound we can put on the expected cost of finding x?

Solution

The expected cost of searching bucket i is $\mathrm{E}\left[\lceil \lg(X_i + 1) \rceil\right]$. This is the expectation of a function of X_i, so we would like to bound it using Jensen's inequality (§4.3).

Unfortunately the function $f(n) = \lceil \lg(n+1) \rceil$ is not concave (because of the ceiling), but $1 + \lg(n+1) > \lceil \lg(n+1) \rceil$ is. So the expected cost of searching bucket i is bounded by $1 + \lg(\mathrm{E}\left[X_i\right] + 1) = 1 + \lg(k+1)$.

Assuming we search the buckets in some fixed order until we find x, we will search Y buckets where $\mathrm{E}\left[Y\right] = \frac{n+1}{2}$. Because Y is determined by the position of x, which is independent of the X_i, Y is also independent of the X_i. So Wald's equation (3.4.2) applies, and the total cost is bounded by

$$\frac{n+1}{2}\left(1 + \lg(k+1)\right).$$

A.3 Assignment 3: due Wednesday, 2014-10-08, at 17:00

A.3.1 Tree contraction

Suppose that you have a tree data structure with n nodes, in which each node i has a pointer $\mathsf{parent}(u)$ to its parent (or itself, in the case of the root node root).

Consider the following randomized algorithm for shrinking paths in the tree: in the first phase, each node u first determines its parent $v = \mathsf{parent}(u)$ and its grandparent $w = \mathsf{parent}(\mathsf{parent}(u))$. In the second phase, it sets $\mathsf{parent}'(u)$ to v or w according to a fair coin-flip independent of the coin-flips of all the other nodes.

Let T be the tree generated by the parent pointers and T' the tree generated by the parent' pointers. An example of two such trees is given in Figure A.1.

Recall that the **depth** of a node is defined by $\mathrm{depth}(\mathsf{root}) = 0$ and $\mathrm{depth}(u) = 1 + \mathrm{depth}(\mathsf{parent}(u))$ when $u \neq \mathsf{root}$. The depth of a tree is equal to the maximum depth of any node.

Let D be depth of T, and D' be the depth of T'. Show that there is a constant a, such that for any fixed $c > 0$, with probability at least $1 - n^{-c}$ it holds that

$$a \cdot D - O\left(\sqrt{D \log n}\right) \leq D' \leq a \cdot D + O\left(\sqrt{D \log n}\right). \tag{A.3.1}$$

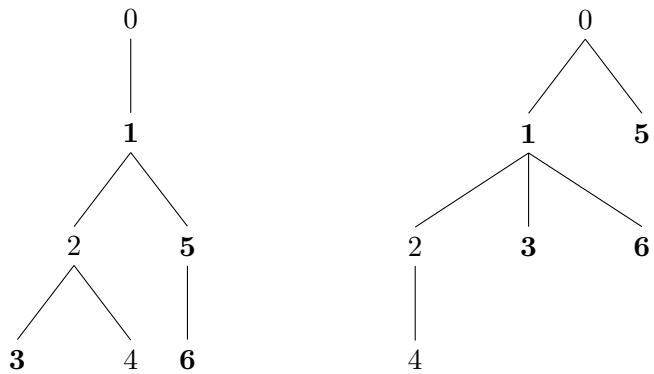


Figure A.1: Example of tree contraction for Problem A.3.1. Tree T is on the left, T' on the right. Nodes 1, 3, 5, and 6 (in boldface) switch to their grandparents. The other nodes retain their original parents.

Solution

For any node u, let $\text{depth}(u)$ be the depth of u in T and $\text{depth}'(u)$ be the depth of u in T'. Note that $\text{depth}'(u)$ is a random variable. We will start by computing $\mathrm{E}\left[\text{depth}'(u)\right]$ as a function of $\text{depth}(u)$, by solving an appropriate recurrence.

Let $S(k) = \text{depth}'(u)$ when $\text{depth}(u) = k$. The base cases are $S(0) = 0$ (the depth of the root never changes) and $S(1) = 1$ (same for the root's children). For larger k, we have

$$\mathrm{E}\left[\text{depth}'(u)\right] = \frac{1}{2}\,\mathrm{E}\left[1 + \text{depth}'(\mathsf{parent}(u))\right] + \frac{1}{2}\,\mathrm{E}\left[1 + \text{depth}'(\mathsf{parent}(\mathsf{parent}(u)))\right]$$

or

$$S(k) = 1 + \frac{1}{2}S(k-1) + \frac{1}{2}S(k-2).$$

There are various ways to solve this recurrence. The most direct may be to define a generating function $F(z) = \sum_{k=0}^{\infty} S(k)z^k$. Then the recurrence becomes

$$F = \frac{z}{1-z} + \frac{1}{2}zF + \frac{1}{2}z^2F.$$

Solving for F gives

$$\begin{aligned}
F &= \frac{\frac{z}{1-z}}{1-\frac{1}{2}z-\frac{1}{2}z^2} \\
&= \frac{2z}{(1-z)(2-z-z^2)}. \\
&= \frac{2z}{(1-z)^2(2+z)} \\
&= 2z\left(\frac{1/3}{(1-z)^2}+\frac{1/9}{(1-z)}+\frac{1/18}{1+\frac{1}{2}z}\right) \\
&= \frac{2}{3}\cdot\frac{z}{(1-z)^2}+\frac{2}{9}\cdot\frac{z}{1-z}+\frac{1}{9}\cdot\frac{z}{1+\frac{1}{2}z},
\end{aligned}$$

from which we can read off the exact solution

$$S(k) = \frac{2}{3}\cdot k + \frac{2}{9} + \frac{1}{9}\left(-\frac{1}{2}\right)^{k-1}$$

when $k \geq 1$.[4]

We can easily show that $\text{depth}'(u)$ is tightly concentrated around $\text{depth}(u)$ using McDiarmid's inequality (5.3.11). Let X_i, for $i = 2\ldots\text{depth}\,u$, be the choice made by u's depth-i ancestor. Then changing one X_i changes $\text{depth}'(u)$ by at most 1. So we get

$$\Pr\left[\text{depth}'(u) \geq \mathrm{E}\left[\text{depth}'(u)+t\right]\right] \leq e^{-2t^2/(\text{depth}(u)-1)} \tag{A.3.2}$$

and similarly

$$\Pr\left[\text{depth}'(u) \leq \mathrm{E}\left[\text{depth}'(u)-t\right]\right] \leq e^{-2t^2/(\text{depth}(u)-1)}. \tag{A.3.3}$$

Let $t = \sqrt{\frac{1}{2}D\ln(1/\epsilon)}$. Then the right-hand side of (A.3.2) and (A.3.3) becomes $e^{-D\ln(1/\epsilon)/(\text{depth}(u)-1)} < e^{-\ln(1/\epsilon)} = \epsilon$. For $\epsilon = \frac{1}{2}n^{-c-1}$, we get $t = \sqrt{\frac{1}{2}D\ln(2n^{c+1})} = \sqrt{\frac{c+1}{2}D(\ln n+\ln 2)} = O\left(\sqrt{D\log n}\right)$ when c is constant.

For the lower bound on D', when can apply (A.3.3) to some single node u with $\text{depth}(u) = D$; this node by itself will give $D' \geq \frac{2}{3}D - O\left(\sqrt{D\log n}\right)$

[4] A less direct but still effective approach is to guess that $S(k)$ grows linearly, and find a and b such that $S(k) \leq ak+b$. For this we need $ak+b \leq 1+\frac{1}{2}(a(k-1)+b)+\frac{1}{2}(a(k-2)+b)$. The b's cancel, leaving $ak \leq 1+ak-\frac{3}{2}a$. Now the ak's cancel, leaving us with $0 \leq 1-\frac{3}{2}a$ or $a \geq 2/3$. We then go back and make $b = 1/3$ to get the right bound on $S(1)$, giving the bound $S(k) \leq \frac{2}{3}\cdot k+\frac{1}{3}$. We can then repeat the argument for $S(k) \geq a'k+b'$ to get a full bound $\frac{2}{3}k \leq S(k) \leq \frac{2}{3}k+\frac{1}{3}$.

with probability at least $1 - \frac{1}{2}n^{-c-1}$. For the upper bound, we need to take the maximum over all nodes. In general, an upper bound on the maximum of a bunch of random variables is likely to be larger than an upper bound on any one of the random variables individually, because there is a lot of room for one of the variables to get unlucky, but we can apply the union bound to get around this. For each individual u, we have $\Pr\left[\text{depth}'(u) \geq \frac{2}{3}D + O\left(\sqrt{D \log n}\right)\right] \leq \frac{1}{2}n^{-c-1}$, so $\Pr\left[D' \geq \frac{2}{3}D + O\left(\sqrt{D \log n}\right)\right] \leq \sum_u \frac{1}{2}n^{-c-1} = \frac{1}{2}n^{-c}$. This completes the proof.

A.3.2 Part testing

You are running a factory that produces parts for some important piece of machinery. Many of these parts are defective, and must be discarded. There are two levels of tests that can be performed:

- A normal test, which passes a part with probability 2/3.
- A rigorous test, which passes a part with probability 1/3.

At each step, the part inspectors apply the following rules to decide which test to apply:

- For the first part, a fair coin-flip decides between the tests.
- For subsequent parts, if the previous part passed, the inspectors become suspicious and apply the rigorous test; if it failed, they relax and apply the normal test.

For example, writing `N+` for a part that passes the normal test, `N-` for one that fails the normal test, `R+` for a part that passes the rigorous test, and `R-` for one that fails the rigorous test, a typical execution of the testing procedure might look like `N- N+ R- N- N+ R- N+ R- N- N- N- N+ R+ R- N+ R+`. This execution tests 16 parts and passes 7 of them.

Suppose that we test n parts. Let S be the number that pass.

1. Compute $\mathrm{E}[S]$.
2. Show that there a constant $c > 0$ such that, for any $t > 0$,

$$\Pr\left[|S - \mathrm{E}[S]| \geq t\right] \leq 2e^{-ct^2/n}. \tag{A.3.4}$$

Solution

Using McDiarmid's inequality and some cleverness Let X_i be the indicator variable for the event that part i passes, so that $S = \sum_{i=1}^{n} X_i$.

1. We can show by induction that $\mathrm{E}[X_i] = 1/2$ for all i. The base case is X_1, where $\Pr[\text{part 1 passes}] = \frac{1}{2}\Pr[\text{part 1 passes rigorous test}] + \frac{1}{2}\Pr[\text{part 1 passes normal test}] = \frac{1}{2}\left(\frac{1}{3} + \frac{2}{3}\right) = \frac{1}{2}$. For $i > 1$, $\mathrm{E}[X_{i-1}] = 1/2$ implies that part i is tested with the normal and rigorous tests with equal probability, so the analysis for X_1 carries through and gives $\mathrm{E}[X_i] = 1/2$ as well. Summing over all X_i gives $\mathrm{E}[S] = n/2$.

2. We can't use Chernoff, Hoeffding, or Azuma here, because the X_i are not independent, and do not form a martingale difference sequence even after centralizing them by subtracting off their expectations. So we are left with McDiarmid's inequality unless we want to do something clever and new (we don't). Applying McDiarmid to the X_i directly doesn't work so well, but we can make it work with a different set of variables that generate the same outcomes.

   Let $Y_i \in \{A, B, C\}$ be the grade of part i, where A means that it passes both the rigorous and the normal test, B means that fails the rigorous test but passes the normal test, and C means that it fails both tests. In terms of the X_i, $Y_i = A$ means $X_i = 1$, $Y_i = C$ means $X_i = 0$, and $Y_i = B$ means $X_i = 1 - X_{i-1}$ (when $i > 1$). We get the right probabilities for passing each test by assigning equal probabilities.

   We can either handle the coin-flip at the beginning by including an extra variable Y_0, or we can combine the coin-flip with Y_1 by assuming that Y_1 is either A or C with equal probability. The latter approach improves our bound a little bit since then we only have n variables and not $n + 1$.

   Now suppose that we fix all Y_j for $j \neq i$ and ask what happens if Y_i changes.

   (a) If $j < i$, then X_j is not affected by Y_i.
   (b) Let $k > i$ be such that $Y_k \in \{A, C\}$. Then X_k is not affected by Y_i, and neither is X_j for any $j > k$.

   It follows that changing Y_i can only change $X_i, \ldots, X_{i+\ell}$, where ℓ is the number of B grades that follow position i.

   There are two cases for the sequence $X_0 \ldots X_\ell$:

(a) If $X_i = 0$, then $X_1 = 1$, $X_2 = 0$, etc.

(b) If $X_i = 1$, then $X_1 = 0$, $X_2 = 1$, etc.

If ℓ is odd, changing Y_i thus has no effect on $\sum_{j=i}^{i+\ell} X_i$, while if ℓ is even, changing Y_i changes the sum by 1. In either case, the effect of changing Y_i is bounded by 1, and McDiarmid's inequality applies with $c_i = 1$, giving

$$\Pr\left[|S - \mathrm{E}\left[S\right]| \geq t\right] \leq 2e^{-2t^2/n}.$$

A.4 Assignment 4: due Wednesday, 2014-10-29, at 17:00

A.4.1 A doubling strategy

A common strategy for keeping the load factor of a hash table down is to double its size whenever it gets too full. Suppose that we start with a hash table of size 1 and double its size whenever two items in a row hash to the same location.

Effectively, this means that we attempt to insert all elements into a table of size 1; if two consecutive items hash to the same location, we start over and try to do the same to a table of size 2, and in general move to a table of size 2^{k+1} whenever any two consecutive elements hash to the same location in our table of size 2^k.

Assume that we are using an independently chosen 2-universal hash function for each table size. Show that the expected final table size is $O(n)$.

Solution

Let X be the random variable representing the final table size. Our goal is to bound $\mathrm{E}\left[X\right]$.

First let's look at the probability that we get at least one collision between consecutive elements when inserting n elements into a table with m locations. Because the pairs of consecutive elements overlap, computing the exact probability that we get a collision is complicated, but we only need an upper bound.

We have $n - 1$ consecutive pairs, and each produces a collision with probability at most $1/m$. This gives a total probability of a collision of at most $(n-1)/m$.

Let $k = \lceil \lg n \rceil$, so that $2^k \geq n$. Then the probability of a consecutive collision in a table with $2^{k+\ell}$ locations is at most $(n-1)/2^{k+\ell} < 2^k/2^{k+\ell} =$

$2^{-\ell}$. Since the events that collisions occur at each table size are independent, we can compute, for $\ell > 0$,

$$\begin{aligned}\Pr\left[X = 2^{k+\ell}\right] &\leq \Pr\left[X \geq 2^{k+\ell}\right] \\ &\leq \prod_{i=0}^{\ell-1} 2^{-i} \\ &= 2^{-\ell(\ell-1)/2}.\end{aligned}$$

From this it follows that

$$\begin{aligned}\mathrm{E}\left[X\right] &= \sum_{i=0}^{\infty} 2^i \Pr\left[X = 2^i\right] \\ &< 2^k \Pr\left[X \leq 2^k\right] + \sum_{l=1}^{\infty} 2^{k+\ell} \Pr\left[X = 2^{k+\ell}\right] \\ &\leq 2^k + \sum_{l=1}^{\infty} 2^{k+\ell} \cdot 2^{-\ell(\ell-1)/2} \\ &\leq 2^k + 2^k \sum_{l=1}^{\infty} 2^{\ell-\ell(\ell-1)/2} \\ &\leq 2^k + 2^k \sum_{l=1}^{\infty} 2^{-(\ell^2-2\ell)/2} \\ &= O(2^k),\end{aligned}$$

since the series converges to some constant that does not depend on k. But we chose k so that $2^k = O(n)$, so this gives us our desired bound.

A.4.2 Hash treaps

Consider the following variant on a treap: instead of choosing the heap keys randomly, we choose a hash function $h : U \to [1 \ldots m]$ from some strongly 2-universal hash family, and use $h(x)$ as the heap key for tree key x. Otherwise the treap operates as usual.[5]

Suppose that $|U| \geq n$. Show that there is a sequence of n distinct tree keys such that the total expected time to insert them into an initially empty hash treap is $\Omega(n^2/m)$.

[5] If m is small, we may get collisions in the heap key values. Assume in this case that a node will stop rising if its parent has the same key.

Solution

Insert the sequence $1 \ldots n$.

Let us first argue by induction on i that all elements x with $h(x) = m$ appear as the uppermost elements of the right spine of the treap. Suppose that this holds for $i - 1$. If $h(i) = m$, then after insertion i is rotated up until it has a parent that also has heap key m; this extends the sequence of elements with heap key m in the spine by 1. Alternatively, if $h(i) < m$, then i is never rotated above an element x with $h(x) = m$, so the sequence of elements with heap key m is unaffected.

Because each new element has a larger tree key than all previous elements, inserting a new element i requires moving past any elements in the right spine, and in particular requires moving past any elements $j < i$ with $h(j) = m$. So the expected cost of inserting i is at least the expected number of such elements j. Because h is chosen from a strongly 2-universal hash family, $\Pr[h(j) = m] = 1/m$ for any j, and by linearity of expectation, $\mathrm{E}[|\{j < i \mid h(j) = m\}|] = (i - 1)/m$. Summing this quantity over all i gives a total expected insertion cost of at least $n(n - 1)/2m = \Omega(n^2/m)$.

A.5 Assignment 5: due Wednesday, 2014-11-12, at 17:00

A.5.1 Agreement on a ring

Consider a ring of n processes at positions $0, 1, \ldots n - 1$. Each process i has a value $A[i]$ that is initially 0 or 1. At each step, we choose a process r uniformly at random, and copy $A[r]$ to $A[(r + 1) \bmod n]$. Eventually, all $A[i]$ will have the same value.

Let $A_t[i]$ be the value of $A[i]$ at time t, and let $X_t = \sum_{i=0}^{n-1} A_t[i]$ be the total number of ones in A_t. Let τ be the first time at which $X_\tau \in \{0, n\}$.

1. Suppose that we start with $X_0 = k$. What is the probability that we eventually reach a state that is all ones?

2. What is $\mathrm{E}[\tau]$, assuming we start from the worst possible initial state?

Solution

This is a job for the optional stopping theorem. Essentially we are going to follow the same analysis from §8.4.1 for a random walk with two absorbing barriers, applied to X_t.

Let $\mathcal{F}_t$ be the σ-algebra generated by $A_0, \ldots, A_t$. Then $\{\mathcal{F}_t\}$ forms a filtration, and each X_t is measurable $\mathcal{F}_t$

1. For the first part, we show that $(X_t, \mathcal{F}_t)$ is a martingale. The intuition is that however the bits in A_t are arranged, there are always exactly the same number of positions where a zero can be replaced by a one as there are where a one can be replaced by a zero.

   Let R_t be the random location chosen in state A_t. Observe that

   $$\mathrm{E}\left[X_{t+1} \mid \mathcal{F}_t, R_t = i\right] = X_t + A[i] - A[(i+1) \bmod n].$$

   But then

   $$\begin{aligned}\mathrm{E}\left[X_{t+1} \mid \mathcal{F}_t\right] &= \sum_{i=0}^{n-1} \frac{1}{n}\left(X_t + A[i] - A[(i+1) \bmod n]\right) \\ &= X_t + \frac{1}{n}X_t - \frac{1}{n}X_t \\ &= X_t,\end{aligned}$$

   which establishes the martingale property.

   We also have that (a) τ is a stopping time with respect to the $\mathcal{F}_i$; (b) $\Pr\left[\tau < \infty\right] = 1$, because from any state there is a nonzero chance that all $A[i]$ are equal n steps later; and (c) X_t is bounded between 0 and n. So the optional stopping theorem applies, giving

   $$\mathrm{E}\left[X_\tau\right] = \mathrm{E}\left[X_0\right] = k.$$

   But then

   $$\mathrm{E}\left[X_\tau\right] = n \cdot \Pr\left[X_\tau = n\right] + 0 \cdot \Pr\left[X_\tau = 0\right] = k,$$

   so $\mathrm{E}\left[X_\tau\right] = k/n$.

2. For the second part, we use a variant on the $X_t^2 - t$ martingale.

   Let Y_t count the number of positions i for which $A_t[i] = 1$ and $A_t[(i+1) \bmod n] = 0$. Then, conditioning on $\mathcal{F}_t$, we have

   $$X_{t+1}^2 = \begin{cases} X_t^2 + 2X_t + 1 & \text{with probability } Y_t/n, \\ X_t^2 - 2X_t + 1 & \text{with probability } Y_t/n, \text{ and} \\ X_t^2 & \text{otherwise.}\end{cases}$$

The conditional expectation sums to

$$\mathrm{E}\left[X_{t+1}^2 \;\middle|\; \mathcal{F}_t\right] = X_t^2 + 2Y_t/n.$$

Let $Z_t = X_t^2 - 2t/n$ when $t \le \tau$ and $X_\tau^2 - 2\tau/n$ otherwise. Then, for $t < \tau$, we have

$$\begin{aligned}
\mathrm{E}\left[Z_{t+1} \mid \mathcal{F}_t\right] &= \mathrm{E}\left[X_{t+1}^2 - 2(t+1)/n \;\middle|\; \mathcal{F}_t\right] \\
&= X_t^2 + 2Y_t/n - 2(t+1)/n \\
&= X_t^2 - 2t/n + 2Y_t/n - 2/n \\
&= Z_t + 2Y_t/n - 2/n \\
&\le Z_t.
\end{aligned}$$

For $t \ge \tau$, $Z_{t+1} = Z_t$, so in either case Z_t has the submartingale property.

We have previously established that τ has bounded expectation, and it's easy to see that Z_t has bounded step size. So the optional stopping theorem applies to Z_t, and $\mathrm{E}\left[Z_0\right] \le \mathrm{E}\left[Z_\tau\right]$.

Let $X_0 = k$. Then $\mathrm{E}\left[Z_0\right] = k^2$, and $\mathrm{E}\left[Z_\tau\right] = (k/n) \cdot n^2 - 2\,\mathrm{E}\left[\tau\right]/n = kn - 2\,\mathrm{E}\left[\tau\right]/n$. But then

$$k^2 \le kn - 2\,\mathrm{E}\left[\tau\right]/n$$

which gives

$$\begin{aligned}
\mathrm{E}\left[\tau\right] &\le \frac{kn - k^2}{2/n} \\
&= \frac{kn^2 - k^2 n}{2}.
\end{aligned}$$

This is maximized at $k = \lfloor n/2 \rfloor$, giving

$$\mathrm{E}\left[\tau\right] \le \frac{\lfloor n/2 \rfloor \cdot n^2 - (\lfloor n/2 \rfloor)^2 \cdot n}{2}.$$

For even n, this is just $n^3/4 - n^3/8 = n^3/8$.

For odd n, this is $(n-1)n^2/4 - (n-1)^2 n/8 = n^3/8 - n/8$. So there is a slight win for odd n from not being able to start with an exact half-and-half split.

To show that this bound applies in the worst case, observe that if we start with have contiguous regions of k ones and $n-k$ zeros in A_t, then (a) $Y_t = 1$, and (b) the two-region property is preserved in A_{t+1}. In this case, for $t < \tau$ it holds that $\mathrm{E}[Z_{t+1} \mid \mathcal{F}_t] = Z_t + 2Y_t/n - 2/n = Z_t$, so Z_t is a martingale, and thus $\mathrm{E}[\tau] = \frac{kn^2-k^2n}{2}$. This shows that the initial state with $\lfloor n/2 \rfloor$ consecutive zeros and $\lceil n/2 \rceil$ consecutive ones (or vice versa) gives the claimed worst-case time.

A.5.2 Shuffling a two-dimensional array

A programmer working under tight time constraints needs to write a procedure to shuffle the elements of an $n \times n$ array, so that all $(n^2)!$ permutations are equally likely. Some quick Googling suggests that this can be reduced to shuffling a one-dimensional array, for which the programmer's favorite language provides a convenient library routine that runs in time linear in the size of the array. Unfortunately, the programmer doesn't read the next paragraph about converting the 2-d array to a 1-d array first, and instead decides to pick one of the $2n$ rows or columns uniformly at random at each step, and call the 1-d shuffler on this row or column.

Let A^t be the state of the array after t steps, where each step shuffles one row or column, and let B be a random variable over permutations of the original array state that has a uniform distribution. Show that the 2-d shuffling procedure above is asympotically worse than the direct approach, by showing that there is some $f(n) = \omega(n)$ such that after $f(n)$ steps, $d_{TV}(A^t, B) = 1 - o(1)$.[6]

Solution

Consider the n diagonal elements in positions A_{ii}. For each such element, there is a $1/n$ chance at each step that its row or column is chosen. The

[6] I've been getting some questions about what this means, so here is an attempt to translate it into English.

Recall that $f(n)$ is $\omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ goes to infinity, and $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ goes to zero.

The problem is asking you to show that there is some $f(n)$ that is more than a constant times n, such that the total variation distance between $A^{f(n)}$ becomes arbitrarily close to 1 for sufficiently large n.

So for example, if you showed that at $t = n^4$, $d_{TV}(A^t, B) \geq 1 - \frac{1}{\log^2 n}$, that would demonstrate the claim, because $\lim_{n\to\infty} \frac{n^2}{n}$ goes to infinity and $\lim_{n\to\infty} \frac{1/\log n}{1} = 0$. These functions are, of course, for illustration only. The actual process might or might not converge by time n^4.)

time until every diagonal node is picked at least once maps to the coupon collector problem, which means that it is $\Omega(n \log n)$ with high probability using standard concentration bounds.

Let C be the event that there is at least one diagonal element that is in its original position. If there is some diagonal node that has not moved, C holds; so with probability $1 - o(1)$, A^t holds at some time t that is $\Theta(n \log n) = \omega(n)$. But by the union bound, C holds in B with probability at most $n \cdot n^{-2} = 1/n$. So the difference between the probability of C in A^t and B is at least $1 - o(1) - 1/n = 1 - o(1)$.

A.6 Assignment 6: due Wednesday, 2014-12-03, at 17:00

A.6.1 Sampling colorings on a cycle

Devise a Las Vegas algorithm for sampling 3-colorings of a cycle.

Given $n > 1$ nodes in a cycle, your algorithm should return a random coloring X of the nodes with the usual constraint that no edge should have the same color on both endpoints. Your algorithm should generate all possible colorings with equal probability, and run in time polynomial in n on average.

Solution

Since it's a Las Vegas algorithm, Markov chain Monte Carlo is not going to help us here. So we can set aside couplings and conductance and just go straight for generating a solution.

First, let's show that we can generate uniform colorings of an n-node line in linear time. Let X_i be the color of node i, where $0 \le i < n$. Choose X_0 uniformly at random from all three colors; then for each $i > 0$, choose X_i uniformly at random from the two colors not chosen for X_{i-1}. Given a coloring, we can show by induction on i that it can be generated by this process, and because each choice is uniform and each coloring is generated only once, we get all $3 \cdot 2^{n-1}$ colorings of the line with equal probability.

Now we try hooking X_{n-1} to X_0. If $X_{n-1} = X_0$, then we don't have a cycle coloring, and have to start over. The probability that this event occurs is at most $1/2$, because for every path coloring with $X_{n-1} = X_0$, there is another coloring where we replace X_{n-1} with a color not equal to X_{n-2} or X_0. So after at most 2 attempts on average we get a good cycle coloring. This gives a total expected cost of $O(n)$.

A.6.2 A hedging problem

Suppose that you own a portfolio of n investment vehicles numbered $1, \ldots n$, where investment i pays out $a_{ij} \in \{-1, +1\}$ at time j, where $0 < j \leq m$. You have carefully chosen these investments so that your total payout $\sum_{i=1}^{n} a_{ij}$ for any time j is zero, eliminating all risk.

Unfortunately, in doing so you have run afoul of securities regulators, who demand that you sell off half of your holdings—a demand that, fortunately, you anticipated by making n even.

This will leave you with a subset S consisting of $n/2$ of the a_i, and your net worth at time t will now be $w_t = \sum_{j=1}^{t} \sum_{i \in S} a_{ij}$. If your net worth drops too low at any time t, all your worldly goods will be repossessed, and you will have nothing but your talent for randomized algorithms to fall back on.

Note: an earlier version of this problem demanded a tighter bound.

Show that when n and m are sufficiently large, it is always possible to choose a subset S of size $n/2$ so that $w_t \geq -m\sqrt{n \ln nm}$ for all $0 < t \leq m$, and give an algorithm that finds such a subset in time polynomial in n and m on average.

Solution

Suppose that we flip a coin independently to choose whether to include each investment i. There are two bad things that can happen:

1. We lose too much at some time t from the investments the coin chooses.

2. We don't get exactly $n/2$ heads.

If we can show that the sum of the probabilities of these bad events is less than 1, we get the existence proof we need. If we can show that it is enough less than 1, we also get an algorithm, because we can test in time $O(nm)$ if a particular choice works.

Let X_i be the indicator variable for the event that we include investment i. Then

$$\begin{aligned}
w_t &= \sum_{i=1}^{n}\left(X_i \sum_{j=1}^{t} a_{ij}\right) \\
&= \sum_{i=1}^{n}\left(\left(\frac{1}{2} + \left(X_i - \frac{1}{2}\right)\right)\sum_{j=1}^{t} a_{ij}\right) \\
&= \frac{1}{2}\sum_{j=1}^{t}\sum_{i=1}^{n} a_{ij} + \sum_{i=1}^{n}\left(X_i - \frac{1}{2}\right)\left(\sum_{i=1}^{t} a_{ij}\right) \\
&= \sum_{i=1}^{n}\left(X_i - \frac{1}{2}\right)\left(\sum_{i=1}^{t} a_{ij}\right).
\end{aligned}$$

Because $\left|X_i - \frac{1}{2}\right|$ is always $\frac{1}{2}$, $\mathrm{E}\left[X_i - \frac{1}{2}\right] = 0$, and each a_{ij} is ± 1, each term in the outermost sum is a zero-mean random variable that satisfies $\left|\left(X_i - \frac{1}{2}\right)\sum_{j=1}^{t} a_{ij}\right| \leq \frac{t}{2} \leq \frac{m}{2}$. So Hoeffding's inequality says

$$\begin{aligned}
\Pr\left[w_t - \mathrm{E}\left[w_t\right] < m\sqrt{n \ln nm}\right] &\leq e^{-m^2 n \ln nm/(2n(m/2)^2)} \\
&= e^{-2\ln nm} \\
&= (nm)^{-2}.
\end{aligned}$$

Summing over all t, the probability that this bound is violated for any t is at most $\frac{1}{n^2 m}$.

For the second source of error, we have $\Pr\left[\sum_{i=1}^{n} X_i \neq n/2\right] = 1 - \binom{n}{2}/2^n = 1 - \Theta(1/\sqrt{n})$. So the total probability that the random assignment fails is bounded by $1 - \Theta(1/\sqrt{n}) + 1/n$, giving a probability that it succeeds of at least $\Theta(1/\sqrt{n}) - 1/(n^2 m) = \Theta(\sqrt{n})$. It follows that generating and testing random assignments gives an assignment with the desired characteristics after $\Theta(\sqrt{n})$ trials on average, giving a total expected cost of $\Theta(n^{3/2} m)$.

A.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

A.7.1 Double records (20 points)

Let $A[1 \ldots n]$ be an array holding the integers $1 \ldots n$ in random order, with all $n!$ permutations of the elements of A equally likely.

Call $A[k]$, where $1 \leq k \leq n$, a **record** if $A[j] < A[k]$ for all $j < k$.

Call $A[k]$, where $2 \leq k \leq n$, a **double record** if both $A[k-1]$ and $A[k]$ are records.

Give an asymptotic (big-Θ) bound on the expected number of double records in A.

Solution

Suppose we condition on a particular set of k elements appearing in positions $A[1]$ through $A[k]$. By symmetry, all $k!$ permutations of these elements are equally likely. Putting the largest two elements in $A[k-1]$ and $A[k]$ leaves $(k-2)!$ choices for the remaining elements, giving a probability of a double record at k of exactly $\frac{(k-2)!}{k!} = \frac{1}{k(k-1)}$.

Applying linearity of expectation gives a total expected number of double records of

$$\begin{aligned}
\sum_{k=2}^{n} \frac{1}{k(k-1)} &\leq \sum_{k=2}^{n} \frac{1}{(k-1)^2} \\
&\leq \sum_{k=2}^{\infty} \frac{1}{(k-1)^2} \\
&= \sum_{k=1}^{\infty} \frac{1}{k^2} \\
&= \frac{\pi^2}{6} \\
&= O(1).
\end{aligned}$$

Since the expected number of double records is at least $1/2 = \Omega(1)$ for $n \geq 2$, this gives a tight asymptotic bound of $\Theta(1)$.

I liked seeing our old friend $\pi^2/6$ so much that I didn't notice an easier

exact bound, which several people supplied in their solutions:

$$\begin{aligned}\sum_{k=2}^{n} \frac{1}{k(k-1)} &= \sum_{k=2}^{n} \left(\frac{1}{k-1} - \frac{1}{k}\right) \\ &= \sum_{k=1}^{n-1} \frac{1}{k} - \sum_{k=2}^{n} \frac{1}{k} \\ &= 1 - \frac{1}{n}. \\ &= O(1).\end{aligned}$$

A.7.2 Hipster graphs (20 points)

Consider the problem of labeling the vertices of a 3-regular graph with labels from $\{0, 1\}$ to maximize the number of *happy* nodes, where a node is *happy* if its label is the opposite of the majority of its three neighbors.

Give a *deterministic* algorithm that takes a 3-regular graph as input and computes, in time polynomial in the size of the graph, a labeling that makes at least half of the nodes happy.

Solution

This problem produced the widest range of solutions, including several very clever deterministic algorithms. Here are some examples.

Using the method of conditional probabilities If we are allowed a randomized algorithm, it's easy to get at exactly half of the nodes on average: simply label each node independently and uniformly at random, observe that each node individually has a 1/2 chance at happiness, and apply linearity of expectation.

To turn this into a deterministic algorithm, we'll apply the method of conditional expectations. Start with an unlabeled graph. At each step, pick a node and assign it a label that maximizes the expected number of happy nodes conditioned on the labeling so far, and assuming that all later nodes are labeled independently and uniformly at random. We can compute this conditional expectation in linear time by computing the value for each node (there are only 3^4 possible partial labelings of the node and its immediate neighbors, so computing the expected happiness of any particular node can be done in constant time by table lookup; to compute the table, we just enumerate all possible assignments to the unlabeled nodes). So in $O(n^2)$

time we get a labeling in which the number of happy nodes is at least the $n/2$ expected happy nodes we started with.

With a bit of care, the cost can be reduced to linear: because each new labeled node only affects its own probability of happiness and those of its three neighbors, we can update the conditional expectation by just updating the values for those four nodes. This gives $O(1)$ cost per step or $O(n)$ total.

Using hill climbing The following algorithm is an adaptation of the solution of Rose Sloan, and demonstrates that it is in fact possible to make *all* of the nodes happy in linear time.

Start with an arbitrary labeling (say, all 0). At each step, choose an unhappy node and flip its label. This reduces the number of monochromatic edges by at least 1. Because we have only $3n/2$ edges, we can repeat this process at most $3n/2$ times before it terminates. But it only terminates when there are no unhappy nodes.

To implement this in linear time, maintain a queue of all unhappy nodes.

A.7.3 Storage allocation (20 points)

Suppose that you are implementing a stack in an array, and you need to figure out how much space to allocate for the array. At time 0, the size of the stack is $X_0 = 0$. At each subsequent time t, the user flips an independent fair coin to choose whether to push onto the stack ($X_t = X_{t-1} + 1$) or pop from the stack ($X_t = X_{t-1} - 1$). The exception is that when the stack is empty, the user always pushes.

You are told in advance that the stack will only be used for n time units. Your job is to choose a size s for the array so that it will overflow at most half the time: $\Pr[\max_t X_t > s] < 1/2$. As an asymptotic (big-$O$) function of n, what is the smallest value of s you can choose?

Solution

We need two ideas here. First, we'll show that $X_t - t^2$ is a martingale, despite the fact that X_t by itself isn't. Second, we'll use the idea of stopping X_t when it hits $s + 1$, creating a new martingale $Y_t = X_{t\wedge\tau}^2 - (t \wedge \tau)$ where τ is the first time where $X_t = s$ (and $t \wedge \tau$ is shorthand for $\min(t, \tau)$). We can then apply Markov's inequality to $X_{n\wedge\tau}$.

To save time, we'll skip directly to showing that Y_t is a martingale. There are two cases:

1. If $t < \tau$, then

$$\begin{aligned}
\mathrm{E}\left[Y_{t+1} \mid Y_t, t < \tau\right] &= \frac{1}{2}\Big((X_t+1)^2 - (t+1)\Big) + \frac{1}{2}\Big((X_t-1)^2 - (t+1)\Big) \\
&= X_t^2 + 1 - (t+1) \\
&= X_t^2 - t \\
&= Y_t.
\end{aligned}$$

2. If $t \geq \tau$, then $Y_{t+1} = Y_t$, so $\mathrm{E}\left[Y_{t+1} \mid Y_t, t \geq \tau\right] = Y_t$.

In either case the martingale property holds.

It follows that $\mathrm{E}\left[Y_n\right] = \mathrm{E}\left[Y_0\right] = 0$, or $\mathrm{E}\left[X_{n\wedge\tau}^2 - n\right] = 0$, giving $\mathrm{E}\left[X_{n\wedge\tau}^2\right] = n$. Now apply Markov's inequality:

$$\begin{aligned}
\Pr\left[\max X_t > s\right] &= \Pr\left[X_{n\wedge\tau} \geq s+1\right] \\
&= \Pr\left[X_{n\wedge\tau}^2 \geq (s+1)^2\right] \\
&\leq \frac{\mathrm{E}\left[X_{n\wedge\tau}^2\right]}{(s+1)^2} \\
&= \frac{n}{(\sqrt{2n}+1)^2} \\
&< \frac{n}{2n} \\
&= 1/2.
\end{aligned}$$

So $s = \sqrt{2n} = O(\sqrt{n})$ is enough.

A.7.4 Fault detectors in a grid (20 points)

A processing plant for rendering discarded final exam problems harmless consists of n^2 processing units arranged in an $n \times n$ grid with coordinates (i, j) each in the range 1 through n. We would like to monitor these processing units to make sure that they are working correctly, and have access to a supply of monitors that will detect failures. Each monitor is placed at some position (i, j), and will detect failures at any of the five positions (i, j), $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, and $(i, j+1)$ that are within the bounds of the grid. This plus-shaped detection range is awkward enough that the engineer designing the system has given up on figuring out how to pack the detectors properly, and instead places a detector at each grid location with independent probability 1/4.

The engineer reasons that since a typical monitor covers 5 grid locations, using $n^2/4$ monitors on average should cover $(5/4)n^2$ locations, with the extra monitors adding a little bit of safety to deal with bad random choices. So few if any processing units should escape.

1. Compute the exact expected number of processing units that are not within range of any monitor, as a function of n. You may assume $n > 0$.

2. Show that for any fixed c, the actual number of unmonitored processing units is within $O(n\sqrt{\log n})$ of the expected number with probability at least $1 - n^{-c}$.

Solution

1. This part is actually more annoying, because we have to deal with nodes on the edges. There are three classes of nodes:

   (a) The four corner nodes. To be unmonitored, each corner node needs to have no monitor in its own location or either of the two adjacent locations. This event occurs with probability $(3/4)^3$.

   (b) The $4(n-2)$ edge nodes. These have three neighbors in the grid, so they are unmonitored with probability $(3/4)^4$.

   (c) The $(n-2)^2$ interior nodes. These are each unmonitored with probability $(3/4)^5$.

   Adding these cases up gives a total expected number of unmonitored nodes of

   $$4 \cdot (3/4)^3 + 4(n-2) \cdot (3/4)^4 + (n-2)^2 \cdot (3/4)^5 = \frac{243}{1024}n^2 - \frac{81}{32}n + \frac{189}{64}. \tag{A.7.1}$$

   For $n = 1$, this analysis breaks down; instead, we can calculate directly that the sole node in the grid has a 3/4 chance of being unmonitored.

   Leaving the left-hand side of (A.7.1) in the original form is probably a good idea for understanding how the result works, but the right-hand side demonstrates that this strategy leaves slightly less than a quarter of the processing units uncovered on average.

2. Let X_{ij} be the indicator for a monitor at position (i, j). Recall that we have assumed that these variables are independent.

Let $S = f(X_{11}, \ldots, X_{nn})$ compute the number of uncovered processing units. Then changing a single X_{ij} changes the value of f by at most 5. So we can apply McDiarmid's inequality to get

$$\Pr\left[|S - \mathrm{E}\left[S\right]| \geq t\right] \leq 2e^{-2t^2/\left(\sum c_{ij}^2\right)} = 2e^{-2t^2/\left(n^2 \cdot 5^2\right)} = 2e^{-2t^2/(25n^2)}.$$

(The actual bound is slightly better, because we are overestimating c_{ij} for boundary nodes.)

Set this to n^{-c} and solve for $t = n\sqrt{(25/2)(c \ln n + \ln 2)} = O(n\sqrt{\log n})$. Then the bound becomes $2e^{-c \ln n - \ln 2} = n^{-c}$, as desired.

Appendix B

Sample assignments from Spring 2013

B.1 Assignment 1: due Wednesday, 2013-01-30, at 17:00

B.1.1 Bureaucratic part

Send me email! My address is `james.aspnes@gmail.com`.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

B.1.2 Balls in bins

Throw m balls independently and uniformly at random into n bins labeled $1 \ldots n$. What is the expected number of positions $i < n$ such that bin i and bin $i + 1$ are both empty?

Solution

If we can figure out the probability that bins i and $i+1$ are both empty for some particular i, then by symmetry and linearity of expectation we can just multiply by $n-1$ to get the full answer.

For bins i and $i+1$ to be empty, every ball must choose another bin. This occurs with probability $(1-2/n)^m$. The full answer is thus $n(1-2/n)^m$, or approximately $ne^{-2m/n}$ when n is large.

B.1.3 A labeled graph

Suppose you are given a graph $G = (V, E)$, where $|V| = n$, and you want to assign labels to each vertex such that the sum of the labels of each vertex and its neighbors modulo $n+1$ is nonzero. Consider the naïve randomized algorithm that assigns a label in the range $0 \ldots n$ to each vertex independently and uniformly at random and then tries again if the resulting labeling doesn't work. Show that this algorithm will find a correct labeling in time polynomial in n on average.

Solution

We'll use the law of total probability. First observe that the probability that a random labeling yields a zero sum for any single vertex and its neighbors is exactly $1/(n+1)$; the easiest way to see this is that after conditioning on the values of the neighbors, there is only one value in $n+1$ that can be assigned to the vertex itself to cause a failure. Now sum this probability over all n vertices to get a probability of failure of at most $n/(n+1)$. It follows that after $n+1$ attempts on average (each of which takes $O(n^2)$ time to check all the neighborhood sums), the algorithm will find a good labeling, giving a total expected time of $O(n^3)$.

B.1.4 Negative progress

An algorithm has the property that if it has already run for n steps, it runs for an additional $n+1$ steps on average. Formally, let $T \geq 0$ be the random variable representing the running time of the algorithm, then

$$\mathrm{E}\left[T \mid T \geq n\right] = 2n + 1. \tag{B.1.1}$$

For each $n \geq 0$, what is the conditional probability $\Pr\left[T = n \mid T \geq n\right]$ that the algorithm stops just after its n-th step?

Solution

Expand (B.1.1) using the definition of conditional expectation to get

$$\begin{aligned}
2n+1 &= \sum_{x=0}^{\infty} x \Pr\left[T = x \mid T \geq n\right] \\
&= \sum_{x=0}^{\infty} x \frac{\Pr\left[T = x \wedge T \geq n\right]}{\Pr\left[T \geq n\right]} \\
&= \frac{1}{\Pr\left[T \geq n\right]} \sum_{x=n}^{\infty} x \Pr\left[T = x\right],
\end{aligned}$$

which we can rearrange to get

$$\sum_{x=n}^{\infty} x \Pr\left[T = x\right] = (2n+1) \Pr\left[T \geq n\right], \tag{B.1.2}$$

provided $\Pr\left[T \geq n\right]$ is nonzero. We can justify this assumption by observing that (a) it holds for $n = 0$, because $T \geq 0$ always; and (b) if there is some $n > 0$ such that $\Pr\left[T \geq n\right] = 0$, then $\mathrm{E}\left[T \mid T \geq n-1\right] = n-1$, contradicting (B.1.1).

Substituting $n+1$ into (B.1.2) and subtracting from the original gives the equation

$$\begin{aligned}
n \Pr\left[T = n\right] &= \sum_{x=n}^{\infty} x \Pr\left[T = x\right] - \sum_{x=n+1}^{\infty} x \Pr\left[T = x\right] \\
&= (2n+1) \Pr\left[T \geq n\right] - (2n+3) \Pr\left[T \geq n+1\right] \\
&= (2n+1) \Pr\left[T = n\right] + (2n+1) \Pr\left[T \geq n+1\right] - (2n+3) \Pr\left[T \geq n+1\right] \\
&= (2n+1) \Pr\left[T = n\right] - 2 \Pr\left[T \geq n+1\right].
\end{aligned}$$

Since we are looking for $\Pr\left[T = n \mid T \geq n\right] = \Pr\left[T = n\right] / \Pr\left[T \geq n\right]$, having an equation involving $\Pr\left[T \geq n+1\right]$ is a bit annoying. But we can borrow a bit of $\Pr\left[T = n\right]$ from the other terms to make it work:

$$\begin{aligned}
n \Pr\left[T = n\right] &= (2n+1) \Pr\left[T = n\right] - 2 \Pr\left[T \geq n+1\right] \\
&= (2n+3) \Pr\left[T = n\right] - 2 \Pr\left[T = n\right] - 2 \Pr\left[T \geq n+1\right] \\
&= (2n+3) \Pr\left[T = n\right] - 2 \Pr\left[T \geq n\right].
\end{aligned}$$

A little bit of algebra turns this into

$$\Pr\left[T = n \mid T \geq n\right] = \frac{\Pr\left[T = n\right]}{\Pr\left[T \geq n\right]} = \frac{2}{n+3}.$$

B.2 Assignment 2: due Thursday, 2013-02-14, at 17:00

B.2.1 A local load-balancing algorithm

Suppose that we are trying to balance n jobs evenly between two machines. Job 1 chooses the left or right machine with equal probability. For $i > 1$, job i chooses the same machine as job $i-1$ with probability p, and chooses the other machine with probability $1-p$. This process continues until every job chooses a machine. We want to estimate how imbalanced the jobs are at the end of this process.

Let X_i be $+1$ if the i-th job chooses the left machine and -1 if it chooses the right. Then $S = \sum_{i=1}^{n} X_i$ is the difference between the number of jobs that choose the left machine and the number that choose the right. By symmetry, the expectation of S is zero. What is the variance of S as a function of p and n?

Solution

To compute the variance, we'll use (5.1.4), which says that $\operatorname{Var}\left[\sum_i X_i\right] = \sum_i \operatorname{Var}[X_i] + 2\sum_{i<j} \operatorname{Cov}[X_i, X_j]$.

Recall that $\operatorname{Cov}[X_i, X_j] = \operatorname{E}[X_i X_j] - \operatorname{E}[X_i]\operatorname{E}[X_j]$. Since the last term is 0 (symmetry again), we just need to figure out $\operatorname{E}[X_i X_j]$ for all $i \le j$ (the $i = j$ case gets us $\operatorname{Var}[X_i]$).

First, let's compute $\operatorname{E}[X_j = 1 \mid X_i = 1]$. It's easiest to do this starting with the $j = i$ case: $\operatorname{E}[X_i \mid X_i = 1] = 1$. For larger j, compute

$$\begin{aligned}\operatorname{E}[X_j \mid X_{j-1}] &= pX_{j-1} + (1-p)(-X_{j-1}) \\ &= (2p-1)X_{j-1}.\end{aligned}$$

It follows that

$$\begin{aligned}\operatorname{E}[X_j \mid X_i = 1] &= \operatorname{E}[(2p-1)X_{j-1} \mid X_i = 1] \\ &= (2p-1)\operatorname{E}[X_{j-1} \mid X_i = 1]. \qquad \text{(B.2.1)}\end{aligned}$$

The solution to this recurrence is $\operatorname{E}[X_j \mid X_i = 1] = (2p-1)^{j-i}$.

We next have

$$\begin{aligned}
\operatorname{Cov}[X_i, X_j] &= \operatorname{E}[X_i X_j] \\
&= \operatorname{E}[X_i X_j \mid X_i = 1] \Pr[X_i = 1] + \operatorname{E}[X_i X_j \mid X_i = -1] \Pr[X_i = -1] \\
&= \frac{1}{2} \operatorname{E}[X_j \mid X_i = 1] + \frac{1}{2} \operatorname{E}[-X_j \mid X_i = -1] \\
&= \frac{1}{2} \operatorname{E}[X_j \mid X_i = 1] + \frac{1}{2} \operatorname{E}[X_j \mid X_i = 1] \\
&= \operatorname{E}[X_j \mid X_i = 1] \\
&= (2p-1)^{j-i}
\end{aligned}$$

as calculated in (B.2.1).

So now we just need to evaluate the horrible sum.

$$\begin{aligned}
\sum_i \operatorname{Var}[X_i] + 2 \sum_{i<j} \operatorname{Cov}[X_i, X_j] &= n + 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} (2p-1)^{j-i} \\
&= n + 2 \sum_{i=1}^{n} \sum_{k=1}^{n-i} (2p-1)^k \\
&= n + 2 \sum_{i=1}^{n} \frac{(2p-1) - (2p-1)^{n-i+1}}{1-(2p-1)} \\
&= n + \frac{n(2p-1)}{1-p} - \frac{1}{1-p} \sum_{m=1}^{n} (2p-1)^m \\
&= n + \frac{n(2p-1)}{1-p} - \frac{(2p-1) - (2p-1)^{n+1}}{2(1-p)^2}.
\end{aligned} \tag{B.2.2}$$

This covers all but the $p = 1$ case, for which the geometric series formula fails. Here we can compute directly that $\operatorname{Var}[S] = n^2$, since S will be $\pm n$ with equal probability.

For smaller values of p, plotting (B.2.2) shows the variance increasing smoothly starting at 0 (for even n) or 1 (for odd n) at $p = 0$ to n^2 in the limit as p goes to 1, with an interesting intermediate case of n at $p = 1/2$, where all terms but the first vanish. This makes a certain intuitive sense: when $p = 0$, the processes alternative which machine they take, which gives an even split for even n and a discrepancy of ± 1 for odd n; when $p = 1/2$, the processes choose machines independently, giving variance n; and for $p = 1$, the processes all choose the same machine, giving n^2.

B.2.2 An assignment problem

Here is an algorithm for placing n balls into n bins. For each ball, we first select two bins uniformly at random without replacement. If at least one of the chosen bins is unoccupied, the ball is placed in the empty bin at no cost. If both chosen bins are occupied, we execute an expensive parallel scan operation to find an empty bin and place the ball there.

1. Compute the exact value of the expected number of scan operations.
2. Let $c > 0$. Show that the absolute value of the difference between the actual number of scan operations and the expected number is at most $O(\sqrt{cn \log n})$ with probability at least $1 - n^{-c}$.

Solution

1. Number the balls 1 to n. For ball i, there are $i - 1$ bins already occupied, giving a probability of $\left(\frac{i-1}{n}\right)\left(\frac{i-2}{n-1}\right)$ that we choose an occupied bin on both attempts and incur a scan. Summing over all i gives us that the expected number of scans is:

$$\begin{aligned}\sum_{i=1}^{n}\left(\frac{i-1}{n}\right)\left(\frac{i-2}{n-1}\right) &= \frac{1}{n(n-1)}\sum_{i=1}^{n-1}(i^2 - i) \\ &= \frac{1}{n(n-1)}\left(\frac{(n-1)n(2n-1)}{6} - \frac{(n-1)n)}{2}\right) \\ &= \frac{(2n-1)}{6} - \frac{1}{2} \\ \frac{n-2}{3},\end{aligned}$$

   provided $n \geq 2$. For $n < 2$, we incur no scans.

2. It's tempting to go after this using Chernoff's inequality, but in this case Hoeffding gives a better bound. Let S be the number of scans. Then S is the sum of n independent Bernoulli random variables, so 5.3.3 says that $\Pr\left[|S - \mathrm{E}\left[S\right]| \geq t\right] \leq 2e^{-2t^2/n}$. Now let $t = \sqrt{cn \ln n} = O(\sqrt{cn \log n})$ to make the right-hand side $2n^{-2c} \leq n^{-c}$ for sufficiently large n.

B.2.3 Detecting excessive collusion

Suppose you have n students in some unnamed Ivy League university's *Introduction to Congress* class, and each generates a random ± 1 vote, with

both outcomes having equal probability. It is expected that members of the same final club will vote together, so it may be that many groups of up to k students each will all vote the same way (flipping a single coin to determine the vote of all students in the group, with the coins for different groups being independent). However, there may also be a much larger conspiracy of exactly m students who all vote the same way (again based on a single independent fair coin), in violation of academic honesty regulations.

Let $c > 0$. How large must m be in asymptotic terms as a function of n, k, and c so that the existence of a conspiracy can be detected solely by looking at the total vote, where the probability of error (either incorrectly claiming a conspiracy when none exists or incorrectly claiming no conspiracy when one exists) is at most n^{-c}?

Solution

Let S be the total vote. The intuition here is that if there is no conspiracy, S is concentrated around 0, and if there is a conspiracy, S is concentrated around $\pm m$. So if m is sufficiently large and $|S| \geq m/2$, we can reasonably guess that there is a conspiracy.

We need to prove two bounds: first, that the probability that we see $|S| \geq m/2$ when there is no conspiracy is small, and second, that the probability that we see $|S| < m/2$ when there is a conspiracy is large.

For the first case, let X_i be the total vote cast by the i-th group. This will be $\pm n_i$ with equal probability, where $n_i \leq k$ is the size of the group. This gives $\mathrm{E}[X_i] = 0$. We also have that $\sum n_i = n$.

Because the X_i are all bounded, we can use Hoeffding's inequality (5.3.1), so long as we can compute an upper bound on $\sum n_i^2$. Here we use the fact that $\sum n_i^2$ is maximized subject to $0 \leq n_i \leq k$ and $\sum n_i = 0$ by setting as many n_i as possible to k; this follows from convexity of $x \mapsto x^2$.[1] We thus have

$$\begin{aligned}\sum n_i^2 &\leq \lfloor n/k \rfloor k^2 + (n \bmod k)^2 \\ &\leq \lfloor n/k \rfloor k^2 + (n \bmod k)k \\ &\leq (n/k)k^2 \\ &= nk.\end{aligned}$$

[1]The easy way to see this is that if f is strictly convex, then f' is increasing. So if $0 < n_i \leq n_j < k$, increasing n_j by ϵ while decreasing n_i by ϵ leaves $\sum n_i$ unchanged while increasing $\sum f(n_i)$ by $\epsilon(f'(n_j) - f'(n_i)) + O(\epsilon^2)$, which will be positive when ϵ is small enough. So at any global maximum, we must have that at least one of n_i or n_j equals 0 or k for any $i \neq j$.

Now apply Hoeffding's inequality to get

$$\Pr\left[|S| \geq m/2\right] \leq 2e^{-(m/2)^2/2nk}.$$

We want to set m so that the right-hand side is less than n^{-c}. Taking logs as usual gives

$$\ln 2 - m^2/8nk \leq -c \ln n,$$

so the desired bound holds when

$$\begin{aligned} m &\geq \sqrt{8nk(c \ln n + \ln 2)} \\ &= \Omega(\sqrt{ckn \log n}). \end{aligned}$$

For the second case, repeat the above analysis on the $n - m$ votes except the $\pm m$ from the conspiracy. Again we get that if $m = \Omega(\sqrt{ckn \log n})$, the probability that these votes exceed $m/2$ is bounded by n^{-c}. So in both cases $m = \Omega(\sqrt{ckn \log n})$ is enough.

B.3 Assignment 3: due Wednesday, 2013-02-27, at 17:00

B.3.1 Going bowling

For your crimes, you are sentenced to play n frames of **bowling**, a game that involves knocking down pins with a heavy ball, which we are mostly interested in because of its complicated scoring system.

In each frame, your result may be any of the integers 0 through 9, a **spare** (marked as `/`), or a **strike** (marked as `X`). We can think of your result on the i-th frame as a random variable X_i. Each result gives a base score B_i, which is equal to X_i when $X_i \in \{0 \ldots 9\}$ and 10 when $X_i \in \{\texttt{/}, \texttt{X}\}$. The actual score Y_i for frame i is the sum of the base scores for the frame and up to two subsequent frames, according to the rule:

$$Y_i = \begin{cases} B_i & \text{when } X_i \in \{0 \ldots 9\}, \\ B_i + B_{i+1} & \text{when } X_i = \texttt{/}, \text{ and} \\ B_i + B_{i+1} + B_{i+2} & \text{when } X_i = \texttt{X}. \end{cases}$$

To ensure that B_{i+2} makes sense even for $i = n$, assume that there exist random variables X_{n+1} and X_{n+2} and the corresponding B_{n+1} and B_{n+2}.

Suppose that the X_i are independent (but not necessarily identically distributed). Show that your final score $S = \sum_{i=1}^{n} Y_i$ is exponentially concentrated[2] around its expected value.

Solution

This is a job for McDiarmid's inequality (5.3.11). Observe that S is a function of $X_1 \ldots X_{n+2}$. We need to show that changing any one of the X_i won't change this function by too much.

From the description of the Y_i, we have that X_i can affect any of Y_{i-2} (if $X_{i-2} = \texttt{X}$), Y_{i-1} (if $X_{i-1} = \texttt{/}$) and Y_i. We can get a crude bound by observing that each Y_i ranges from 0 to 30, so changing X_i can change $\sum Y_i$ by at most ± 90, giving $c_i \leq 90$. A better bound can be obtained by observing that X_i contributes only B_i to each of Y_{i-2} and Y_{i-1}, so changing X_i can only change these values by up to 10; this gives $c_i \leq 50$. An even more pedantic bound can be obtained by observing that X_1, X_2, X_{n+1}, and X_{n+2} are all special cases, with $c_1 = 30$, $c_2 = 40$, $c_{n+1} = 20$, and $c_{n+2} = 10$, respectively; these values can be obtained by detailed meditation on the rules above.

We thus have $\sum_{i=1}^{n+2} c_i^2 = (n-2)50^2 + 30^2 + 40^2 + 20^2 + 10^2 = 2500(n-2) + 3000 = 2500n - 2000$, assuming $n \geq 2$. This gives $\Pr\left[|S - \mathrm{E}\left[S\right]| \geq t\right] \leq \exp(-2t^2/(2500n - 2000))$, with the symmetric bound holding on the other side as well.

For the standard game of bowling, with $n = 10$, this bound starts to bite at $t = \sqrt{11500} \approx 107$, which is more than a third of the range between the minimum and maximum possible scores. There's a lot of variance in bowling, but this looks like a pretty loose bound for players who don't throw a lot of strikes. For large n, we get the usual bound of $O\left(\sqrt{n \log n}\right)$ with high probability: the averaging power of endless repetition eventually overcomes any slop in the constants.

B.3.2 Unbalanced treaps

Recall that a **treap** (§6.3) is only likely to be balanced if the sequence of insert and delete operations applied to it is independent of the priorities chosen by the algorithm.

Suppose that we insert the keys 1 through n into a treap with random priorities as usual, but then allow the adversary to selectively delete

[2] This means "more concentrated than you can show using Chebyshev's inequality."

whichever keys it wants to after observing the priorities assigned to each key.

Show that there is an adversary strategy that produces a path in the treap after deletions that has expected length $\Omega(\sqrt{n})$.

Solution

An easy way to do this is to produce a tree that consists of a single path, which we can do by arranging that the remaining keys have priorities that are ordered the same as their key values.

Here's a simple strategy that works. Divide the keys into $\sqrt{n}$ ranges of $\sqrt{n}$ keys each ($1 \ldots \sqrt{n}$, $\sqrt{n+1} \ldots 2\sqrt{n}$, etc.).[3] Rank the priorities from 1 to n. From each range $(i-1)\sqrt{n} \ldots i\sqrt{n}$, choose a key to keep whose priority is also ranked in the range $(i-1)\sqrt{n} \ldots i\sqrt{n}$ (if there is one), or choose no key (if there isn't). Delete all the other keys.

For a particular range, we are drawing $\sqrt{n}$ samples without replacement from the n priorities, and there are $\sqrt{n}$ possible choices that cause us to keep a key in that range. The probability that every draw misses is $\prod_{i=1}^{\sqrt{n}}(1 - \sqrt{n}/(n-i+1)) \leq (1 - 1/\sqrt{n})^{\sqrt{n}} \leq e^{-1}$. So each range contributes at least $1 - e^{-1}$ keys on average. Summing over all $\sqrt{n}$ ranges gives a sequence of keys with increasing priorities with expected length at least $(1 - e^{-1})\sqrt{n} = \Omega(\sqrt{n})$.

An alternative solution is to apply the **Erdős-Szekeres theorem** [ES35], which says that every sequence of length k^2+1 has either an increasing subsequence of length $k+1$ or a decreasing sequence of $k+1$. Consider the sequence of priorities corresponding to the keys $1 \ldots n$; letting $k = \lfloor \sqrt{n-1} \rfloor$ gives a subsequence of length at least $\sqrt{n-1}$ that is either increasing or decreasing. If we delete all other elements of the treap, the elements corresponding to this subsequence will form a path, giving the desired bound. Note that this does not require any probabilistic reasoning at all.

Though not required for the problem, it's possible to show that $\Theta(\sqrt{n})$ is the best possible bound here. The idea is that the number of possible sequences of keys that correspond to a path of length k in a binary search tree is exactly $\binom{n}{k}2^{k-1}$; the $\binom{n}{k}$ corresponds to choosing the keys in the path, and the 2^{k-1} is because for each node except the last, it must contain either the smallest or the largest of the remaining keys because of the binary search tree property.

[3] To make our life easer, we'll assume that n is a square. This doesn't affect the asymptotic result.

```
        *
       / \
    a /   \ b
     /     \
    *       *
   / \       \
a /   \ b     \ b
 /     \       \
aa     ab      bb
```

Figure B.1: A radix tree, storing the strings `aa`, `ab`, and `ba`.

Since each such sequence will be a treap path only if the priorities are decreasing (with probability $1/k!$), the union bound says that the probability of having any length-k paths is at most $\binom{n}{k}2^{k-1}/k!$. But

$$\begin{aligned}\binom{n}{k}2^{k-1}/k! &\leq \frac{(2n)^k}{2(k!)^2} \\ &\geq \frac{(2n)^k}{2(k/e)^{2k}} \\ &= \frac{1}{2}(2e^2n/k^2)^k.\end{aligned}$$

This is exponentially small for $k \gg \sqrt{2e^2n}$, showing that with high probability all possible paths have length $O(\sqrt{n})$.

B.3.3 Random radix trees

A **radix tree** over an alphabet of size m is a tree data structure where each node has up to m children, each corresponding to one of the letters in the alphabet. A string is represented by a node at the end of a path whose edges are labeled with the letters in the string in order. For example, in Figure B.1, the string `ab` is stored at the node reached by following the `a` edge out of the root, then the `b` edge out of this child.

The only nodes created in the radix tree are those corresponding to stored keys or ancestors of stored keys.

Suppose you have a radix tree into which you have already inserted n strings of length k from an alphabet of size m, generated uniformly at random with replacement. What is the expected number of new nodes you need to create to insert a new string of length k?

Solution

We need to create a new node for each prefix of the new string that is not already represented in the tree.

For a prefix of length ℓ, the chance that none of the n strings have this prefix is exactly $\left(1 - m^{-\ell}\right)^n$. Summing over all ℓ gives that the expected number of new nodes is $\sum_{\ell=0}^{k} \left(1 - m^{-\ell}\right)^n$.

There is no particularly clean expression for this, but we can observe that $(1 - m^{-\ell})^n \leq \exp(-nm^{-\ell})$ is close to zero for $\ell < \log_m n$ and close to 1 for $\ell > \log_m n$. This suggests that the expected value is $k - \log_m n + O(1)$.

B.4 Assignment 4: due Wednesday, 2013-03-27, at 17:00

B.4.1 Flajolet-Martin sketches with deletion

A **Flajolet-Martin sketch** [FNM85] is a streaming data structure for approximately counting the number of distinct items n in a large data stream using only $m = O(\log n)$ bits of storage.[4] The idea is to use a hash function h that generates each value $i \in \{1, \ldots m\}$ with probability 2^{-i}, and for each element x that arrives in the data stream, we write a 1 to $A[h(x)]$. (With probability 2^{-m} we get a value outside this range and write nowhere.) After inserting n distinct elements, we estimate n as $\hat{n} = 2^{\hat{k}}$, where $\hat{k} = \max\{k \mid A[k] = 1\}$, and argue that this is likely to be reasonably close to n.

Suppose that we modify the Flajolet-Martin sketch to allow an element x to be deleted by writing 0 to $A[h(x)]$. After n insertions and d deletions (of distinct elements in both cases), we estimate the number of remaining elements $n - d$ as before by $\widehat{n - d} = 2^{\hat{k}}$, where $\hat{k} = \max\{k \mid A[k] = 1\}$.

Assume that we never delete an element that has not previously been inserted, and that the values of h are for different inputs are independent of each other and of the sequence of insertions and deletions.

Show that there exist constants $c > 1$ and $\epsilon > 0$, such that for n sufficiently large, after inserting n distinct elements then deleting $d \leq \epsilon n$ of them, $\Pr\left[(n-d)/c \leq \widehat{n-d} \leq (n-d)c\right] \geq 2/3$.

[4] If you actually need to do this, there exist better data structures for this problem. See [KNW10].

Solution

We'll apply the usual error budget approach and show that the probability that $\widehat{n-d}$ is too big and the probability that $\widehat{n-d}$ is too small are both small. For the moment, we will leave c and ϵ as variables, and find values that work at the end.

Let's start with the too-big side. To get $A[k] = 1$, we need $h(x_i) = k$ for some x_i that is inserted but not subsequently deleted. There are $n - d$ such x_i, and each gives $h(x_i) = k$ with probability 2^{-k}. So $\Pr[A[k] = 1] \leq (n-d)2^{-k}$. This gives

$$\begin{aligned}
\Pr\left[\widehat{n-d} \geq (n-d)c\right] &= \Pr\left[\widehat{k} \geq \lceil \lg((n-d)c)\rceil\right] \\
&\leq \sum_{k=\lceil \lg((n-d)c)\rceil}^{\infty} (n-d)2^{-k} \\
&= 2(n-d)2^{-\lceil \lg((n-d)c)\rceil} \\
&\leq \frac{2}{c}.
\end{aligned}$$

On the too-small side, fix $k = \lceil \lg((n-d)/c))\rceil$. Since $A[k] = 1$ gives $\hat{k} \geq k \geq \lceil \lg((n-d)/c))\rceil$, we have $\Pr\left[\widehat{n-d} < (n-d)/c\right] = \Pr\left[\hat{k} < \lg(n-d)/c\right] \leq \Pr[A[k] = 0]$. (We might be able to get a better bound by looking at larger indices, but to solve the problem this one k will turn out to be enough.)

Let $x_1 \ldots x_{n-d}$ be the values that are inserted and not later deleted, and $x_{n-d+1} \ldots x_n$ the values that are inserted and then deleted. For $A[k]$ to be zero, either (a) no x_i for i in $1 \ldots x_{n-d}$ has $h(x_i) = k$; or (b) some x_i for i in $n - d + 1 \ldots x_n$ has $h(x_i) = k$. The probability of the first event is

$\left(1-2^{-k}\right)^{n-d}$; the probability of the second is $1-\left(1-2^{-k}\right)^{d}$. So we have

$$\begin{aligned}
\Pr\left[A[k]=0\right] &\leq \left(1-2^{-k}\right)^{n-d} + \left(1-\left(1-2^{-k}\right)^{d}\right) \\
&\leq \exp\left(-2^{-k}(n-d)\right) + \left(1-\exp\left(-\left(2^{-k}+2^{-2k}\right)d\right)\right) \\
&\leq \exp\left(-2^{-\lceil \lg((n-d)/c))\rceil}(n-d)\right) + \left(1-\exp\left(-2\cdot 2^{-\lceil \lg((n-d)/c))\rceil}d\right)\right) \\
&\leq \exp\left(-2^{-\lg((n-d)/c))}(n-d)\right) + \left(1-\exp\left(-2\cdot 2^{-\lg((n-d)/c))+1}d\right)\right) \\
&= e^{-c} + \left(1-\exp\left(-\frac{4cd}{n-d}\right)\right) \\
&\leq e^{-c} + \frac{4cd}{n-d} \\
&\leq e^{-c} + \frac{4c\epsilon}{1-\epsilon}.
\end{aligned}$$

So our total probability of error is bounded by $\frac{2}{c}+e^{-c}+\frac{4c\epsilon}{1-\epsilon}$. Let $c=8$ and $\epsilon=1/128$ to make this less than $\frac{1}{4}+e^{-8}+\frac{128}{127}\cdot\frac{1}{16}\approx 0.313328<1/3$, giving the desired bound.

B.4.2 An adaptive hash table

Suppose we want to build a hash table, but we don't know how many elements we are going to put in it, and because we allow undisciplined C programmers to obtain pointers directly to our hash table entries, we can't move an element once we assign it a position to it. Here we will consider a data structure that attempts to solve this problem.

Construct a sequence of tables $T_0, T_1, \ldots,$ where each T_i has $m_i = 2^{2^i}$ slots. For each table T_i, choose k independent strongly 2-universal hash functions $h_{i1}, h_{i2}, \ldots h_{ik}$.

The insertion procedure is given in Algorithm B.1. The essentially idea is that we make k attempts (each with a different hash function) to fit x into T_0, then k attempts to fit it in T_1, and so on.

If the tables T_i are allocated only when needed, the space complexity of this data structure is given by the sum of m_i for all tables that have at least one element.

Show that for any fixed $\epsilon > 0$, there is a constant k such that after inserting n elements:

1. The expected cost of an additional insertion is $O(\log\log n)$, and
2. The expected space complexity is $O\left(n^{2+\epsilon}\right)$.

```
procedure insert(x)
    for i ← 0 to ∞ do
        for j ← 1 to k do
            if T_i[h_ij(x)] = ⊥ then
                T_i[h_ij(x)] ← x
                return
```

Algorithm B.1: Adaptive hash table insertion

Solution

The idea is that we use T_{i+1} only if we get a collision in T_i. Let X_i be the indicator for the event that there is a collision in T_i. Then

$$\mathrm{E}\left[\text{steps}\right] \leq 1 + \sum_{i=0}^{\infty} \mathrm{E}\left[X_i\right] \tag{B.4.1}$$

and

$$\mathrm{E}\left[\text{space}\right] \leq m_0 + \sum_{i=0}^{\infty} \mathrm{E}\left[X_i\right] m_{i+1}. \tag{B.4.2}$$

To bound $\mathrm{E}\left[X_i\right]$, let's calculate an upper bound on the probability that a newly-inserted element x_{n+1} collides with any of the previous n elements $x_1 \ldots x_n$ in table T_i. This occurs if, for every location $h_{ij}(x_{n+1})$, there is some x_r and some j' such that $h_{ij'}(x_r) = h_{ij}(x_{n+1})$. The chance that this occurs for any particular j, j', and r is at most $1/m_i$ (if $j = j'$, use 2-universality of h_{ij}, and if $j \neq j'$, use independence and uniformity), giving a chance that it occurs for fixed j that is at most n/m_i. The chance that it occurs for all j is at most $(n/m_i)^k$, and the expected number of such collisions summed over any $n+1$ elements that we insert is bounded by $n(n/m_i)^k$ (the first element can't collide with any previous elements). So we have $\mathrm{E}\left[X_i\right] \leq \min\left(1, n(n/m_i)^k\right)$.

Let ℓ be the largest value such that $m_\ell \leq n^{2+\epsilon}$. We will show that, for an appropriate choice of k, we are sufficiently unlikely to get a collision in round ℓ that the right-hand sides of (B.4.2) and (B.4.2) end up being not much more than the corresponding sums up to $\ell - 1$.

From our choice of ℓ, it follows that (a) $\ell \leq \lg\lg n^{2+\epsilon} = \lg\lg n + \lg(2+\epsilon) = O(\log\log n)$; and (b) $m_{\ell+1} > n^{2+\epsilon}$, giving $m_\ell = \sqrt{m_{\ell+1}} > n^{1+\epsilon/2}$. From this we get $\mathrm{E}\left[X_\ell\right] \leq n(n/m_\ell)^k < n^{1-k\epsilon/2}$.

By choosing k large enough, we can make this an arbitrarily small polynomial in n. Our goal is to wipe out the $\mathrm{E}[X_\ell]\, m_{\ell+1}$ and subsequent terms in (B.4.2).

Observe that $\mathrm{E}[X_i]\, m_{i+1} \le n(n/m_i)^k m_i^2 = n^{k+1} m_i^{2-k}$. Let's choose k so that this is at most $1/m_i$, when $i \ge \ell$, so we get a nice convergent series.[5] This requires $n^{k+1} m_i^{3-k} \le 1$ or $k + 1 + (3-k)\log_n m_i \le 0$. If $i \ge \ell$, we have $\log_n m_i > 1 + \epsilon/2$, so we win if $k + 1 + (3-k)(1+\epsilon/2) \le 0$. Setting $k \ge 8/\epsilon + 3$ works. (Since we want k to be an integer, we probably want $k = \lceil 8/\epsilon \rceil + 3$ instead.)

So now we have

$$\begin{aligned}
\mathrm{E}[\text{space}] &\le m_0 + \sum_{i=0}^{\infty} \mathrm{E}[X_i]\, m_{i+1} \\
&\le \sum_{i=0}^{\ell} m_i + \sum_{i=\ell}^{\infty} \frac{1}{m_i} \\
&\le 2m_\ell + \frac{2}{m_\ell} \\
&= O(n^{2+\epsilon}).
\end{aligned}$$

For $\mathrm{E}[\text{steps}]$, compute the same sum without all the m_{i+1} factors. This makes the tail terms even smaller, so they is still bounded by a constant, and the head becomes just $\sum_{i=0}^{\ell} 1 = O(\log\log n)$.

B.4.3 An odd locality-sensitive hash function

A deranged computer scientist decides that if taking one bit from a random index in a bit vector is a good way to do locality-sensitive hashing (see §7.7.1.3), then taking the exclusive OR of k independently chosen indices must be even better.

Formally, given a bit-vector $x_1 x_2 \ldots x_n$, and a sequence of indices $i_1 i_2 \ldots i_k$, define $h_i(x) = \bigoplus_{j=1}^{k} x_{i_j}$. For example, if $x = 00101$ and $i = 3, 5, 2$, $h_i(x) = 1 \oplus 1 \oplus 0 = 0$.

Suppose x and y are bit-vectors of length n that differ in m places.

1. Give a *closed-form expression* for the probability that $h_i(x) \ne h_i(y)$, assuming i consists of k indices chosen uniformly and independently at random from $1 \ldots n$

[5] We could pick a smaller k, but why not make things easier for ourselves?

2. Use this to compute the exact probability that $h_i(x) \neq h_i(y)$ when $m = 0$, $m = n/2$, and $m = n$.

Hint: You may find it helpful to use the identity $(a \bmod 2) = \frac{1}{2}(1 - (-1)^a)$.

Solution

1. Observe that $h_i(x) \neq h_i(y)$ if an only if i chooses an odd number of indices where x and y differ. Let $p = m/n$ be the probability that each index in i hits a position where x and y differ, and let $q = 1 - p$. Then the event that we get an odd number of differences is

$$\begin{aligned}
\sum_{j=0}^{k} (j \bmod 2) \binom{k}{j} p^j q^{k-j} &= \sum_{j=0}^{k} \frac{1}{2}\left(1 - (-1)^j\right) \binom{k}{j} p^j q^{k-j} \\
&= \frac{1}{2} \sum_{j=0}^{k} \binom{k}{j} p^j q^{k-j} - \frac{1}{2} \sum_{j=0}^{k} \binom{k}{j} (-p)^j q^{k-j} \\
&= \frac{1}{2}(p+q)^k - \frac{1}{2}(-p+q)^k \\
&= \frac{1 - (1 - 2(m/n))^k}{2}.
\end{aligned}$$

2.
   - For $m = 0$, this is $\frac{1-1^k}{2} = 0$.
   - For $m = n/2$, it's $\frac{1-0^k}{2} = \frac{1}{2}$ (assuming $k > 0$).
   - For $m = n$, it's $\frac{1-(-1)^k}{2} = (k \bmod 2)$.

In fact, the chances of not colliding as a function of m are symmetric around $m = n/2$ if k is even and increasing if k is odd. So we can only hope to use this as locality-sensitive hash function in the odd case.

B.5 Assignment 5: due Friday, 2013-04-12, at 17:00

B.5.1 Choosing a random direction

Consider the following algorithm for choosing a random direction in three dimensions. Start at the point $(X_0, Y_0, Z_0) = (0, 0, 0)$. At each step, pick one of the three coordinates uniformly at random and add ± 1 to it with equal probability. Continue until the resulting vector has length at least k, i.e., until $X_t^2 + Y_t^2 + Z_t^2 \geq k^2$. Return this vector.

What is the expected running time of this algorithm, as an asymptotic function of k?

Solution

The trick is to observe that $X_t^2 + Y_t^2 + Z_t^2 - t$ is a martingale, essentially following the same analysis as for $X_t^2 - t$ for a one-dimensional random walk. Suppose we pick X_t to change. Then

$$\begin{aligned} \mathrm{E}\left[X_{t+1}^2 \;\middle|\; X_t\right] &= \frac{1}{2}\left((X_t+1)^2 + (X_t-1)^2\right) \\ &= X_t^2 + 1. \end{aligned}$$

So

$$\mathrm{E}\left[X_{t+1}^2 + Y_{t+1}^2 + Z_{t+1}^2 - (t+1) \;\middle|\; X_t, Y_t, Z_t, X \text{ changes}\right] = X_t^2 + Y_t^2 + Z_t^2 - t.$$

But by symmetry, the same equation holds if we condition on Y or Z changing. It follows that $\mathrm{E}\left[X_{t+1}^2 + Y_{t+1}^2 + Z_{t+1}^2 - (t+1) \mid X_t, Y_t, Z_t\right] = X_t^2 + Y_t^2 + Z_t^2 - t$, and that we have a martingale as claimed.

Let τ be the first time at which $X_t^2 + Y_t^2 + Z_t^2 \leq k^2$. From the optional stopping theorem (specifically, the bounded-increments case of Theorem 8.2.1), $\mathrm{E}\left[X_\tau^2 + Y_\tau^2 + Z_\tau^2 - \tau\right] = 0$, or equivalently $\mathrm{E}[\tau] = \mathrm{E}\left[X_\tau^2 + Y_\tau^2 + Z_\tau^2\right]$. This immediately gives $\mathrm{E}[\tau] \geq k^2$.

To get an upper bound, observe that $X_{\tau-1}^2 + Y_{\tau-1}^2 + Z_{\tau-1}^2 < k^2$, and that exactly one of these three term increases between $\tau - 1$ and τ. Suppose it's X (the other cases are symmetric). Increasing X by 1 sets $X_\tau^2 = X_{\tau-1}^2 + 2X_{\tau-1} + 1$. So we get

$$\begin{aligned} X_\tau^2 + Y_\tau^2 + Z_\tau^2 &= \left(X_{\tau-1}^2 + Y_{\tau-1}^2 + Z_{\tau-1}^2\right) + 2X_{\tau-1} + 1 \\ &< k^2 + 2k + 1. \end{aligned}$$

So we have $k^2 \leq \mathrm{E}[\tau] < k^2 + 2k + 1$, giving $\mathrm{E}[\tau] = \Theta(k^2)$. (Or $k^2 + O(k)$ if we are feeling really precise.)

B.5.2 Random walk on a tree

Consider the following random walk on a (possibly unbalanced) binary search tree: At each step, with probability 1/3 each, move to the current node's parent, left child, or right child. If the target node does not exist, stay put.

Suppose we adapt this random walk using Metropolis-Hastings (see §9.3.3) so that the probability of each node at depth d in the stationary distribution is proportional to α^{-d}.

Use a coupling argument to show that, for any constant $\alpha > 2$, this adapted random walk converges in $O(D)$ steps, where D is the depth of the tree.

Solution

As usual, let X_t be a copy of the chain starting in an arbitrary initial state and Y_t be a copy starting in the stationary distribution.

From the Metropolis-Hastings algorithm, the probability that the walk moves to a particular child is $1/3\alpha$, so the probability that the depth increases after one step is at most $2/3\alpha$. The probability that the walk moves to the parent (if we are not already at the root) is $1/3$.

We'll use the same choice (left, right, or parent) in both the X and Y processes, but it may be that only one of the particles moves (because the target node doesn't exist). To show convergence, we'll track $Z_t = \max(\text{depth}(X_t), \text{depth}(Y_t))$. When $Z_t = 0$, both X_t and Y_t are the root node.

There are two ways that Z_t can change:

1. Both processes choose "parent"; if Z_t is not already 0, $Z_{t+1} = Z_t - 1$. This case occurs with probability $1/3$.

2. Both processes choose one of the child directions. If the appropriate child exists for the deeper process (or for either process if they are at the same depth), we get $Z_{t+1} = Z_t + 1$. This even occurs with probability at most $2/3\alpha < 1/3$.

So the expected change in Z_{t+1} conditioned on $Z_t > 0$ is at most $-1/3 + 2/3\alpha = -(1/3)(2/\alpha - 1)$. Let τ be the first time at which $Z_t = 0$. Then the process $Z'_t = Z_t - (1/3)(2/\alpha - 1)t$ for $t \leq \tau$ and 0 for $t > \tau$ is a supermartingale, so $\mathrm{E}[Z_\tau] = \mathrm{E}[Z_0] = \mathrm{E}[\max(\text{depth}(X_0), \text{depth}(Y_0))] \leq D$. This gives $\mathrm{E}[\tau] \leq \frac{3D}{2/\alpha - 1}$.

B.5.3 Sampling from a tree

Suppose we want to sample from the stationary distribution of the Metropolis-Hastings walk in the previous problem, but we don't want to do an actual random walk. Assuming $\alpha > 2$ is a constant, give an algorithm for sampling *exactly* from the stationary distribution that runs in constant expected time.

Your algorithm should not require knowledge of the structure of the tree. Its only input should be a pointer to the root node.

Clarification added 2013-04-09: Your algorithm can determine the children of any node that it has already found. The idea of not knowing the structure of the tree is that it can't, for example, assume a known bound on the depth, counts of the number of nodes in subtrees, etc., without searching through the tree to find this information directly.

Solution

We'll use rejection sampling. The idea is to choose a node in the infinite binary tree with probability proportional to α^{-d}, and then repeat the process if we picked a node that doesn't actually exist. Conditioned on finding a node i that exists, its probability will be $\frac{\alpha^{\operatorname{depth}(x)}}{\sum_j \alpha^{-\operatorname{depth}(j)}}$.

If we think of a node in the infinite tree as indexed by a binary strength of length equal to its depth, we can generate it by first choosing the length X and then choosing the bits in the string. We want $\Pr[X = n]$ to be proportional to $2^n \alpha^{-n} = (2/\alpha)^n$. Summing the geometric series gives

$$\Pr[X = n] = \frac{(2/\alpha)^n}{1 - (2/\alpha)}.$$

This is a geometric distribution, so we can sample it by repeatedly flipping a biased coin. The number of such coin-flips is $O(X)$, as is the number of random bits we need to generate and the number of tree nodes we need to check. The *expected* value of X is given by the infinite series

$$\sum_{n=0}^{\infty} \frac{(2/\alpha)^n n}{1 - (2/\alpha)};$$

this series converges to some constant by the ratio test.

So each probe costs $O(1)$ time on average, and has at least a constant probability of success, since we choose the root with $\Pr[X = 0] = \frac{1}{1-2/\alpha}$. Using Wald's equation (8.4.1), the total expected time to run the algorithm is $O(1)$.

This is a little surprising, since the output of the algorithm may have more than constant length. But we are interested in expectation, and when $\alpha > 2$ most of the weight lands near the top of the tree.

B.6 Assignment 6: due Friday, 2013-04-26, at 17:00

B.6.1 Increasing subsequences

Let $S_1, \ldots, S_m$ be sets of indices in the range $1 \ldots n$. Say that a permutation π of $1 \ldots n$ is increasing on S_j if $\pi(i_1) < \pi(i_2) < \cdots < \pi(i_{k_j})$ where $i_1 < i_2 < \cdots < i_{k_j}$ are the elements of S_j.

Given a fully polynomial-time randomized approximation scheme that takes as input n and a sequence of sets $S_1, \ldots, S_m$, and approximates the number of permutations π that are increasing on at least one of the S_j.

Solution

This can be solved using a fairly straightforward application of Karp-Luby [KL85] (see §10.3). Recall that for Karp-Luby we need to be able to express our target set U as the union of a polynomial number of covering sets U_j, where we can both compute the size of each U_j and sample uniformly from it. We can then estimate $|U| = \sum_{j,x \in U_j} f(j,x) = \left(\sum_j |U_j|\right) \Pr\left[f(j,x) = 1\right]$ where $f(j,x)$ is the indicator for the event that $x \notin U_{j'}$ for any $j' < j$ and in the probability, the pair (j,x) is chosen uniformly at random from $\{(j,x) \mid x \in U_j\}$.

In this case, let U_j be the set of all permutations that are increasing on S_j. We can specify each such permutation by specifying the choice of which $k_j = |S_j|$ elements are in positions $i_1 \ldots i_{k_j}$ (the order of these elements is determined by the requirement that the permutation be increasing on S_j) and specifying the order of the remaining $n - k_j$ elements. This gives $\binom{n}{k_j}(n-k_j)! = (n)_{n-k_j}$ such permutations. Begin by computing these counts for all S_j, as well as their sum.

We now wish to sample uniformly from pairs (j, π) where each π is an element of S_j. First sample each j with probability $|S_j| / \sum_\ell |S_\ell|$, using the counts we've already computed. Sampling a permutation uniformly from S_j mirrors the counting argument: choose a k_j-subset for the positions in S_j, then order the remaining elements randomly. The entire sampling step can easily be done in $O(n+m)$ time.

Computing $f(j,\pi)$ requires testing π to see if it is increasing for any $S_{j'}$ for $j < j'$; without doing anything particularly intelligent, this takes $O(nm)$ time. So we can construct and test one sample in $O(nm)$ time. Since each sample has at least a $\rho = 1/m$ chance of having $f(j,\pi) = 1$, from Lemma 10.2.1 we need $O\left(\frac{1}{\epsilon^2 \rho} \log \frac{1}{\delta}\right) = O\left(m\epsilon^{-2} \log \frac{1}{\delta}\right)$ samples to get relative error with probability at least $1-\delta$, for a total cost of $O\left(m^2 n \epsilon^{-2} \log \frac{1}{\delta}\right)$.

```
bacab  bacab
ccaac  ccaac
bbbac  babac
bbaaa  bbaaa
acbab  acbab
```

Figure B.2: Non-futile (left) and futile (right) word search grids for the lexicon {`aabc`, `ccca`}

B.6.2 Futile word searches

A **word search puzzle** consists of an $n \times n$ grid of letters from some alphabet Σ, where the goal is to find contiguous sequences of letters in one of the eight orthogonal or diagonal directions that form words from some lexicon. For example, in Figure B.2, the left grid contains an instance of `aabc` (running up and left from the rightmost `a` character on the last line), while the right grid contains no instances of this word.

For this problem, you are asked to build an algorithm for constructing word search puzzles with no solution for a given lexicon. That is, given a set of words S over some alphabet and a grid size n, the output should be an $n \times n$ grid of letters such that no word in S appears as a contiguous sequence of letters in one of the eight directions anywhere in the grid. We will refer to such puzzles as **futile word search puzzles**.

1. Suppose the maximum length of any word in S is k. Let p_S be the probability that some word in S is a prefix of an infinite string generated by picking letters uniformly and independently from Σ. Show that there is a constant $c > 0$ such that for any k, Σ, and S, $p_S < ck^{-2}$ implies that there exists, for all n, an $n \times n$ futile word search puzzle for S using only letters from Σ.

2. Give an algorithm that constructs a futile word search puzzle given S and n in expected time polynomial in $|S|$, k, and n, provided $p_S < ck^{-2}$ as above.

Solution

1. We'll apply the symmetric version of the Lovász local lemma. Suppose the grid is filled in independently and uniformly at random with characters from Σ. Given a position ij in the grid, let A_{ij} be the

event that there exists a word in S whose first character is at position ij; observe that $\Pr[A_{ij}] \leq 8p_S$ by the union bound (this may be an overestimate, both because we might run off the grid in some directions and because the choice of initial character is not independent). Observe also that A_{ij} is independent of any event $A_{i'j'}$ where $|i - i'| \geq 2k - 1$ or $|j - j'| \geq 2k - 1$, because no two words starting at these positions can overlap. So we can build a dependency graph with $p \leq 8p_S$ and $d \leq (4k - 3)^2$. The Lovász local lemma shows that there exists an assignment where no A_{ij} occurs provided $ep(d + 1) < 1$ or $8ep_S((4k - 3)^2 + 1) < 1$. This easily holds if $p_S < \frac{1}{8e(4k^2)} = \frac{1}{128e}k^{-2}$.

2. For this part, we can just use Moser-Tardos [MT10], particularly the symmetric version described in Corollary 11.3.4. We have a collection of $m = O(n^2)$ bad events, with $d = \Theta(k^2)$, so the expected number of resamplings is bounded by $m/d = O(n^2/k^2)$. Each resampling requires checking every position in the new grid for an occurrence of some string in S; this takes $O(n^2k \cdot |S|)$ time per resampling even if we are not very clever about it. So the total expected cost is $O(n^4 \cdot |S|/k)$.

   With some more intelligence, this can be improved. We don't need to recheck any position at distance greater than k from any of the at most k letters we resample, and if we are sensible, we can store S using a radix tree or some similar data structure that allows us to look up all words that appear as a prefix of a given length-k string in time $O(k)$. This reduces the cost of each resampling to $O(k^3)$, with an additive cost of $O(k \cdot |S|)$ to initialize the data structure. So the total expected cost is now $O(n^2k + |S|)$.

B.6.3 Balance of power

Suppose you are given a set of n MMORPG players, and a sequence of subsets $S_1, S_2, \ldots, S_m$ of this set, where each subset S_i gives the players who will participate in some raid. Before any of the raids take place, you are to assign each player permanently to one of three factions. If for any i, $|S_i|/2$ or more of the players are in the same faction, then instead of carrying out the raid they will overwhelm and rob the other participants.

Give a randomized algorithm for computing a faction assignment that prevents this tragedy from occurring (for all i) and thus allows all m raids to be completed without incident, assuming that $m > 1$ and $\min_i |S_i| \geq c \ln m$ for some constant $c > 0$ that does not depend on n or m. Your algorithm should run in expected time polynomial in n and m.

Solution

Assign each player randomly to a faction. Let X_{ij} be the number of players in S_i that are assigned to faction j. Then $\mathrm{E}[X_{ij}] = |S_i|/3$. Applying the Chernoff bound (5.2.2), we have

$$\begin{aligned}\Pr[X_{ij} \geq |S_i|/2] &= \Pr\left[X_{ij} \geq \left(1+\frac{1}{2}\right)\mathrm{E}[X_{ij}]\right] \\ &\leq \exp\left(-(|S_i|/3)\left(\frac{1}{2}\right)^2/3\right) \\ &= e^{-|S_i|/36}.\end{aligned}$$

Let $c = 3 \cdot 36 = 108$. Then if $\min_i |S_i| \geq c \ln m$, for each i, j, it holds that $\Pr[X_{ij} \geq |S_i|/2] \leq e^{-3\ln m} = m^{-3}$. So the probability that this bound is exceeded for any i and j is at most $(3m)m^{-2} = 3/m^2$. So a random assignment works with at least $1/4$ probability for $m > 1$.

We can generate and test each assignment in $O(nm)$ time. So our expected time is $O(nm)$.

B.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

B.7.1 Dominating sets

A **dominating set** in a graph is a subset S of the vertices for which every vertex v is either in S or adjacent to a vertex in S.

Show that for any graph G, there is an aperiodic, irreducible Markov chain on the dominating sets of G, such that (a) the transition rule for the chain can be implemented in polynomial time; and (b) the stationary distribution of the chain is uniform. (You do not need to say anything about the convergence rate of this chain.)

Solution

Suppose we are have state S_t at time t. We will use a random walk where we choose a vertex uniformly at random to add to or remove from S_t, and carry out the action only if the resulting set is still a dominating set.

In more detail: For each vertex v, with probability $1/n$, $S_{t+1} = S_t \cup \{v\}$ if $v \notin S_t$, $S_{t+1} = S_t \setminus \{v\}$ if $v \in S_t$ and $S_t \setminus \{v\}$ is a dominating set, and $S_{t+1} = S_t$ otherwise. To implement this transition rule, we need to be able to choose a vertex v uniformly at random (easy) and test in the case where $v \in S_t$ if $S_t \setminus \{v\}$ is a dominating set (also polynomial: for each vertex, check if it or one of its neighbors is in $S_t \setminus \{v\}$, which takes time $O(|V| + |E|)$). Note that we do not need to check if $S_t \cup \{v\}$ is a dominating set.

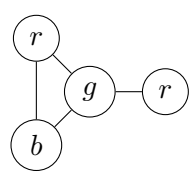
For any pair of adjacent states S and $S' = S \setminus \{v\}$ the probability of moving from S to S' and the probability of moving from S' to S are both $1/n$. So the Markov chain is reversible with a uniform stationary distribution.

This is an aperiodic chain, because there exist minimal dominating sets for which there is a nonzero chance that $S_{t+1} = S_t$.

It is irreducible, because for any dominating set S, there is a path to the complete set of vertices V by adding each vertex in $V \setminus S$ one at a time. Conversely, removing these vertices from V gives a path to S. This gives a path $S \rightsquigarrow V \rightsquigarrow T$ between any two dominating sets S and T.

B.7.2 Tricolor triangles

Suppose you wish to count the number of assignments of colors $\{r, g, b\}$ to nodes of a graph G that have the property that some triangle in G contains all three colors. For example, the four-vertex graph shown below is labeled with one of 18 such colorings (6 permutations of the colors of the triangle nodes times 3 unconstrained choices for the degree-1 node).



Give a fully polynomial-time randomized approximation scheme for this problem.

Solution

Though it is possible, and tempting, to go after this using Karp-Luby (see §10.3), naive sampling is enough.

If a graph has at least one triangle (which can be checked in $O(n^3)$ time just by enumerating all possible triangles), then the probability that that particular triangle is tricolored when colors are chosen uniformly and

independently at random is $6/27 = 2/9$. This gives a constant hit rate ρ, so by Lemma 10.2.1, we can get ϵ relative error with $1 - \delta$ probability using $O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta}\right)$ samples. Each sample costs $O(n^3)$ time to evaluate (again, brute-force checking of all possible triangles), for a total cost of $O\left(n^3 \epsilon^{-2} \log \frac{1}{\delta}\right)$.

B.7.3 The n rooks problem

The n rooks problem requires marking as large a subset as possible of the squares in an $n \times n$ grid, so that no two squares in the same row or column are marked.[6]

Consider the following randomized algorithm that attempts to solve this problem:

1. Give each of the n^2 squares a distinct label using a uniformly chosen random permutation of the integers $1 \ldots n^2$.

2. Mark any square whose label is larger than any other label in its row and column.

What is the expected number of marked squares?

Solution

Each square is marked if it is the largest of the $2n - 1$ total squares in its row and column. By symmetry, each of these $2n - 1$ squares is equally likely to be the largest, so the probability that a particular square is marked is exactly $\frac{1}{2n-1}$. By linearity of expectation, the total expected number of marked squares is then $\frac{n^2}{2n-1}$.

B.7.4 Pursuing an invisible target on a ring

Suppose that you start at position 0 on a ring of size $2n$, while a target particle starts at position n. At each step, starting at position i, you can choose whether to move to any of positions $i - 1$, i, or $i + 1$. At the same time, the target moves from its position j to either $j - 1$ or $j + 1$ with equal probability, independent of its previous moves or your moves. Aside from knowing that the target starts at n at time 0, you cannot tell where the target moves.

[6] This is not actually a hard problem.

Your goal is to end up on the same node as the target after some step.[7] Give an algorithm for choosing your moves such that, for any $c > 0$, you encounter the target in at most $2n$ steps with probability at least $1 - n^{-c}$ for sufficiently large n.

Solution

The basic idea is to just go through positions $0, 1, 2, \ldots$ until we encounter the target, but we have to be a little careful about parity to make sure we don't pass it by accident.[8]

Let $X_i = \pm 1$ be the increment of the target's i-th move, and let $S_i = \sum_{j=1}^{i} X_j$, so that its position after i steps is $n + S_i \bmod 2n$.

Let Y_i be the position of the pursuer after i steps.

First move: stay at 0 if n is odd, move to 1 if n is even. The purpose of this is to establish the invariant that $n + S_i - Y_i$ is even starting at $i = 1$. For subsequent moves, let $Y_{i+1} = Y_i + 1$. Observe that this maintains the invariant.

We assume that n is at least 2. This is necessary to ensure that at time 1, $Y_1 \leq n + S_1$.

Claim: if at time $2n$, $Y_{2n} \geq n + S_{2n}$, then at some time $i \leq 2n$, $Y_i = n + S_i$. Proof: Let i be the first time at which $Y_i \geq n + S_i$; under the assumption that $n \geq 2$, $i \geq 1$. So from the invariant, we can't have $Y_i = n + S_i + 1$, and if $Y_i \geq n + S_i + 2$, we have $Y_{i-1} \geq Y_i - 1 \geq n + S_i + 1 \geq n + S_{i-1}$, contradicting our assumption that i is minimal. The remaining alternative is that $Y_i = n + S_i$, giving a collision at time i.

We now argue that $Y_{2n} \geq n - 1$ is very likely to be at least $n + S_{2n}$. Since S_{2n} is a sum of $2n$ independent ± 1 variables, from Hoeffding's inequality we

[7] Note that you must be in the same place at the end of the step: if you move from 1 to 2 while the target moves from 2 to 1, that doesn't count.

[8] This is not the only possible algorithm, but there are a lot of plausible-looking algorithms that turn out not to work. One particularly tempting approach is to run to position n using the first n steps and then spend the next n steps trying to hit the target in the immediate neighborhood of n, either by staying put (a sensible strategy when lost in the woods in real life, assuming somebody is looking for you), or moving in a random walk of some sort starting at n. This doesn't work if we want a high-probability bound. To see this, observe that the target has a small but nonzero constant probability in the limit of begin at some position greater than or equal to $n + 4\sqrt{n}$ after exactly $n/2$ steps. Conditioned on starting at $n + 4\sqrt{n}$ or above, its chances of moving below $n + 4\sqrt{n} - 2\sqrt{n} = n + 2\sqrt{n}$ at any time in the next $3n/2$ steps is bounded by $e^{-4n/2(3n/2)} = e^{-4/3}$ (Azuma), and a similar bound holds independently for our chances of getting up to $n + 2\sqrt{n}$ or above. Multiplying out all these constants gives a constant probability of failure. A similar but bigger disaster occurs if we don't rush to n first.

have $\Pr[Y_n < n + S_{2n}] \leq \Pr[S_{2n} \geq n] \leq e^{-n^2/4n} = e^{-n/4}$. For sufficiently large n, this is much smaller than n^{-c} for any fixed c.

Appendix C

Sample assignments from Spring 2011

C.1 Assignment 1: due Wednesday, 2011-01-26, at 17:00

C.1.1 Bureaucratic part

Send me email! My address is `james.aspnes@gmail.com`.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

C.1.2 Rolling a die

The usual model of a randomized algorithm assumes a source of fair, independent random bits. This makes it easy to generate uniform numbers in the range $0 \ldots 2^n - 1$, but not so easy for other ranges. Here are two algorithms for generating a uniform random integer $0 \leq s < n$:

- **Rejection sampling** generates a uniform random integer $0 \leq s < 2^{\lceil \lg n \rceil}$. If $s < n$, return s; otherwise keep trying until you get some $s < n$.

- **Arithmetic coding** or **range coding** generates a sequence of bits $r_1, r_2, \ldots r_k$ until the half-open interval $[\sum_{i=1}^{k} 2^{-i} r_i, \sum_{i=1}^{k} 2^{-i} r_i + 2^{-k-1})$ is a subset of $[s/n, (s+1)/n)$ for some s; it then returns s.

1. Show that both rejection sampling and range coding produce a uniform value $0 \leq s < n$ using an expected $O(\log n)$ random bits.

2. Which algorithm has a better constant?

3. Does there exist a function f and an algorithm that produces a uniform value $0 \leq s < n$ for any n using $f(n)$ random bits with probability 1?

Solution

1. For rejection sampling, each sample requires $\lceil \lg n \rceil$ bits and is accepted with probability $n/2^{\lceil \lg n \rceil} \geq 1/2$. So rejection sampling returns a value after at most 2 samples on average, using no more than an expected $2 \lceil \lg n \rceil < 2(\lg n + 1)$ expected bits for the worst n.

   For range coding, we keep going as long as one of the $n-1$ nonzero endpoints s/n lies inside the current interval. After k bits, the probability that one of the 2^k intervals contains an endpoint is at most $(n-1)2^{-k}$; in particular, it drops below 1 as soon as $k = 2^{\lceil \lg n \rceil}$ and continues to drop by 1/2 for each additional bit, requiring 2 more bits on average. So the expected cost of range coding is at most $\lceil \lg n \rceil + 2 < \lg n + 3$ bits.

2. We've just shown that range coding beats rejection sampling by a factor of 2 in the limit, for worst-case n. It's worth noting that other factors might be more important if random bits are cheap: rejection sampling is much easier to code and avoids the need for division.

3. There is no algorithm that produces a uniform value $0 \leq s < n$ for all n using any fixed number of bits. Suppose such an algorithm existed. Fix some n. For all n values s to be equally likely, the sets of random bits $M^{-1}(s) = \{r \mid M(r) = s\}$ must have the same size. But this can only happen if n divides $2^{f(n)}$, which works only for n a power of 2.

C.1.3 Rolling many dice

Suppose you repeatedly roll an n-sided die. Give an asymptotic (big-Θ) bound on the expected number of rolls until you roll some number you have already rolled before.

Solution

In principle, it is possible to compute this value exactly, but we are lazy.

For a lower bound, observe that after m rolls, each of the $\binom{m}{2}$ pairs of rolls has probability $1/n$ of being equal, for an expected total of $\binom{m}{2}/n$ duplicates. For $m = \sqrt{n}/2$, this is less than $1/8$, which shows that the expected number of rolls is $\Omega(\sqrt{n})$.

For the upper bound, suppose we have already rolled the die $\sqrt{n}$ times. If we haven't gotten a duplicate already, each new roll has probability at least $\sqrt{n}/n = 1/\sqrt{n}$ of matching a previous roll. So after an additional $\sqrt{n}$ rolls on average, we get a repeat. This shows that the expected number of rolls is $O(\sqrt{n})$.

Combining these bounds shows that we need $\Theta(\sqrt{n})$ rolls on average.

C.1.4 All must have candy

A set of n_0 children each reach for one of n_0 candies, with each child choosing a candy independently and uniformly at random. If a candy is chosen by exactly one child, the candy and child drop out. The remaining n_1 children and candies then repeat the process for another round, leaving n_2 remaining children and candies, etc. The process continues until ever child has a candy.

Give the best bound you can on the expected number of rounds until every child has a candy.

Solution

Let $T(n)$ be the expected number of rounds remaining given we are starting with n candies. We can set up a probabilistic recurrence relation $T(n) = 1 + T(n - X_n)$ where X_n is the number of candies chosen by eactly one child. It is easy to compute $\mathrm{E}[X_n]$, since the probability that any candy gets chosen exactly once is $n(1/n)(1 - 1/n)^{n-1} = (1 - 1/n)^{n-1}$. Summing over all candies gives $\mathrm{E}[X_n] = n(1 - 1/n)^{n-1}$.

The term $(1-1/n)^{n-1}$ approaches e^{-1} in the limit, so for any fixed $\epsilon > 0$, we have $n(1 - 1/n)^{n-1} \geq n(e^{-1} - \epsilon)$ for sufficiently large n. We can get a quick bound by choosing ϵ so that $e^{-1} - \epsilon \geq 1/4$ (for example) and then

applying the Karp-Upfal-Wigderson inequality (E.3.1) with $\mu(n) = n/4$ to get

$$\begin{aligned}\mathrm{E}\left[T(n)\right] &\leq \int_1^n \frac{1}{t/4}\,dt \\ &= 4\ln n.\end{aligned}$$

There is a sneaky trick here, which is that we stop if we get down to 1 candy instead of 0. This avoids the usual problem with KUW and $\ln 0$, by observing that we can't ever get down to exactly one candy: if there were exactly one candy that gets grabbed twice or not at all, then there must be some other candy that also gets grabbed twice or not at all.

This analysis is sufficient for an asymptotic estimate: the last candy gets grabbed in $O(\log n)$ rounds on average. For most computer-sciency purposes, we'd be done here.

We can improve the constant slightly by observing that $(1 - 1/n)^{n-1}$ is in fact always greater than or equal to e^{-1}. The easiest way to see this is to plot the function, but if we want to prove it formally we can show that $(1 - 1/n)^{n-1}$ is a decreasing function by taking the derivative of its logarithm:

$$\begin{aligned}\frac{d}{dn}\ln(1-1/n)^{n-1} &= \frac{d}{dn}(n-1)\ln(1-1/n) \\ &= \ln(1-1/n) + \frac{n-1}{1-1/n}\cdot\frac{-1}{n^2}.\end{aligned}$$

and observing that it is negative for $n > 1$ (we could also take the derivative of the original function, but it's less obvious that it's negative). So if it approaches e^{-1} in the limit, it must do so from above, implying $(1-1/n)^{n-1} \geq e^{-1}$.

This lets us apply (E.3.1) with $\mu(n) = n/e$, giving $\mathrm{E}\left[T(n)\right] \leq e\ln n$.

If we skip the KUW bound and use the analysis in §E.4.2 instead, we get that $\Pr\left[T(n) \geq \ln n + \ln(1/\epsilon)\right] \leq \epsilon$. This suggests that the actual expected value should be $(1 + o(1))\ln n$.

C.2 Assignment 2: due Wednesday, 2011-02-09, at 17:00

C.2.1 Randomized dominating set

A **dominating set** in a graph $G = (V, E)$ is a set of vertices D such that each of the n vertices in V is either in D or adjacent to a vertex in D.

Suppose we have a d-regular graph, in which every vertex has exactly d neighbors. Let D_1 be a random subset of V in which each vertex appears with independent probability p. Let D be the union of D_1 and the set of all vertices that are not adjacent to any vertex in D_1. (This construction is related to a classic maximal independent set algorithm of Luby [Lub85], and has the desirable property in a distributed system of finding a dominating set in only one round of communication.)

1. What would be a good value of p if our goal is to minimize $\mathrm{E}[|D|]$, and what bound on $\mathrm{E}[|D|]$ does this value give?

2. For your choice of p above, what bound can you get on $\Pr[|D| - \mathrm{E}[|D|] \geq t]$?

Solution

1. First let's compute $\mathrm{E}[|D|]$. Let X_v be the indicator for the event that $v \in D$. Then $X_v = 1$ if either (a) v is in D_1, which occurs with probability p; or (b) v and all d of its neighbors are not in D_1, which occurs with probability $(1-p)^{d+1}$. Adding these two cases gives $\mathrm{E}[X_v] = p + (1-p)^{d+1}$ and thus

$$\mathrm{E}[|D|] = \sum_v \mathrm{E}[X_v] = n\left(p + (1-p)^{d+1}\right). \tag{C.2.1}$$

   We optimize $\mathrm{E}[|D|]$ in the usual way, by seeking a minimum for $\mathrm{E}[X_v]$. Differentiating with respect to p and setting to 0 gives $1-(d+1)(1-p)^d = 0$, which we can solve to get $p = 1-(d+1)^{-1/d}$. (We can easily observe that this must be a minimum because setting p to either 0 or 1 gives $\mathrm{E}[X_v] = 1$.)

   The value of $\mathrm{E}[|D|]$ for this value of p is the rather nasty expression $n(1-(d+1)^{-1/d} + (d+1)^{-1-1/d})$.

   Plotting the d factor up suggests that it goes to $\ln d/d$ in the limit, and both Maxima and `www.wolframalpha.com` agree with this. Knowing

the answer, we can prove it by showing

$$\begin{aligned}
\lim_{d\to\infty} \frac{1-(d+1)^{-1/d}+(d+1)^{-1-1/d}}{\ln d/d} &= \lim_{d\to\infty} \frac{1-d^{-1/d}+d^{-1-1/d}}{\ln d/d} \\
&= \lim_{d\to\infty} \frac{1-d^{-1/d}}{\ln d/d} \\
&= \lim_{d\to\infty} \frac{1-e^{-\ln d/d}}{\ln d/d} \\
&= \lim_{d\to\infty} \frac{1-\left(1-\ln d/d+O(\ln^2 d/d^2)\right)}{\ln d/d} \\
&= \lim_{d\to\infty} \frac{\ln d/d+O(\ln^2 d/d^2)}{\ln d/d} \\
&= \lim_{d\to\infty} (1+O(\ln d/d)) \\
&= 1.
\end{aligned}$$

This lets us write $\mathrm{E}[|D|] = (1+o(1))n \ln d/d$, where we bear in mind that the $o(1)$ term depends on d but not n.

2. Suppose we fix X_v for all but one vertex u. Changing X_u from 0 to 1 can increase $|D|$ by at most one (if u wasn't already in D) and can decrease it by at most $d-1$ (if u wasn't already in D and adding u to D_1 lets all d of u's neighbors drop out). So we can apply the method of bounded differences with $c_i = d-1$ to get

$$\Pr[|D| - \mathrm{E}[|D|] \geq t] \leq \exp\left(-\frac{t^2}{2n(d-1)^2}\right).$$

A curious feature of this bound is that it doesn't depend on p at all. It may be possible to get a tighter bound using a better analysis, which might pay off for very large d (say, $d \gg \sqrt{n}$).

C.2.2 Chernoff bounds with variable probabilities

Let $X_1 \ldots X_n$ be a sequence of Bernoulli random variables, where for all i, $\mathrm{E}[X_i \mid X_1 \ldots X_{i-1}] \leq p_i$. Let $S = \sum_{i=1}^n X_i$ and $\mu = \sum_{i=1}^n p_i$. Show that, for all $\delta \geq 0$,

$$\Pr[S \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu.$$

Solution

Let $S_t = \sum_{i=1}^{t} X_i$, so that $S = S_n$, and let $\mu_t = \sum_{i=1}^{t} p_i$. We'll show by induction on t that $\mathrm{E}\left[e^{\alpha S_t}\right] \leq \exp(e^{\alpha-1}\mu_t)$, when $\alpha > 0$.

Compute

$$\begin{aligned}
\mathrm{E}\left[e^{\alpha S}\right] &= \mathrm{E}\left[e^{\alpha S_{n-1}} e^{\alpha X_n}\right] \\
&= \mathrm{E}\left[e^{\alpha S_{n-1}} \mathrm{E}\left[e^{\alpha X_n} \;\middle|\; X_1, \ldots, X_{n-1}\right]\right] \\
&= \mathrm{E}\left[e^{\alpha S_{n-1}} \left(\Pr\left[X_n = 0 \mid X_1, \ldots, X_{n-1}\right] + e^{\alpha} \Pr\left[X_n = 1 \mid X_1, \ldots, X_{n-1}\right]\right)\right] \\
&= \mathrm{E}\left[e^{\alpha S_{n-1}} \left(1 + (e^{\alpha} - 1) \Pr\left[X_n = 1 \mid X_1, \ldots, X_{n-1}\right]\right)\right] \\
&\leq \mathrm{E}\left[e^{\alpha S_{n-1}} \left(1 + (e^{\alpha} - 1) p_n\right)\right] \\
&\leq \mathrm{E}\left[e^{\alpha S_{n-1}} \exp\left((e^{\alpha} - 1) p_n\right)\right] \\
&\leq \mathrm{E}\left[e^{\alpha S_{n-1}}\right] \exp\left((e^{\alpha} - 1) p_n\right) \\
&\leq \exp\left(e^{\alpha} - 1)\mu_{n-1}\right) \exp\left((e^{\alpha} - 1) p_n\right) \\
&= \exp\left(e^{\alpha} - 1)\mu_n\right).
\end{aligned}$$

Now apply the rest of the proof of (5.2.1) to get the full result.

C.2.3 Long runs

Let W be a binary string of length n, generated uniformly at random. Define a **run** of ones as a maximal sequence of contiguous ones; for example, the string 11100110011111101011 contains 5 runs of ones, of length 3, 2, 6, 1, and 2.

Let X_k be the number of runs in W of length k or more.

1. Compute the exact value of $\mathrm{E}\left[X_k\right]$ as a function of n and k.
2. Give the best concentration bound you can for $|X_k - \mathrm{E}\left[X_k\right]|$.

Solution

1. We'll compute the probability that any particular position $i = 1 \ldots n$ is the start of a run of length k or more, then sum over all i. For a run of length k to start at position i, either (a) $i = 1$ and $W_i \ldots W_{i+k-1}$ are all 1, or (b) $i > 1$, $W_{i-1} = 0$, and $W_i \ldots W_{i+k-1}$ are all 1. Assuming $n \geq k$, case (a) adds 2^{-k} to $\mathrm{E}\left[X_k\right]$ and case (b) adds $(n-k)2^{-k-1}$, for a total of $2^{-k} + (n-k)2^{-k-1} = (n-k+2)2^{-k-1}$.

2. We can get an easy bound without too much cleverness using McDiarmid's inequality (5.3.11). Observe that X_k is a function of the independent random variables $W_1 \dots W_n$ and that changing one of these bits changes X_k by at most 1 (this can happen in several ways: a previous run of length $k-1$ can become a run of length k or vice versa, or two runs of length k or more separated by a single zero may become a single run, or vice versa). So (5.3.11) gives $\Pr\left[|X - \mathrm{E}\left[X\right]| \geq t\right] \leq 2\exp\left(-\frac{t^2}{2n}\right)$.

   We can improve on this a bit by grouping the W_i together into blocks of length ℓ. If we are given control over a block of ℓ consecutive bits and want to minimize the number of runs, we can either (a) make all the bits zero, causing no runs to appear within the block and preventing adjacent runs from extending to length k using bits from the block, or (b) make all the bits one, possibly creating a new run but possibly also causing two existing runs on either side of the block to merge into one. In the first case, changing all the bits to one except for a zero after every k consecutive ones creates at most $\left\lfloor \frac{\ell+2k-1}{k+1} \right\rfloor$ new runs. Treating each of the $\lceil n \rceil \ell$ blocks as a single variable then gives $\Pr\left[|X - \mathrm{E}\left[X\right]| \geq t\right] \leq 2\exp\left(-\frac{t^2}{2\lceil n/\ell \rceil (\lfloor (\ell+2k-1)/(k+1) \rfloor)^2}\right)$. Staring at plots of the denominator for a while suggests that it is minimized at $\ell = k+3$, the largest value with $\lfloor (\ell+2k-1)/(k+1) \rfloor \leq 2$. This gives $\Pr\left[|X - \mathrm{E}\left[X\right]| \geq t\right] \leq 2\exp\left(-\frac{t^2}{8\lceil n/(k+3) \rceil}\right)$, improving the bound on t from $\Theta(\sqrt{n\log(1/\epsilon)})$ to $\Theta(\sqrt{(n/k)\log(1/\epsilon)})$.

   For large k, the expectation of any individual X_k becomes small, so we'd expect that Chernoff bounds would work better on the upper bound side than the method of bounded differences. Unfortunately, we don't have independence. But from Problem C.2.2, we know that the usual Chernoff bound works as long as we can show $\mathrm{E}\left[X_i \mid X_1, \dots, X_{i-1}\right] \leq p_i$ for some sequence of fixed bounds p_i.

   For X_1, there are no previous X_i, and we have $\mathrm{E}\left[X_1\right] = 2^{-k}$ exactly.

   For X_i with $i > 1$, fix $X_1, \dots, X_{i-1}$; that is, condition on the event $X_j = x_j$ for all $j < i$ with some fixed sequence $x_1, \dots, x_{i-1}$. Let's call this event A. Depending on the particular values of the x_j, it's not clear how conditioning on A will affect X_i; but we can split on the

value of W_{i-1} to show that either it has no effect or $X_i = 0$:

$$\begin{aligned}\mathrm{E}[X_i \mid A] &= \mathrm{E}[X_i \mid A, W_{i-1} = 0]\Pr[W_{i-1} = 0 \mid A] + \mathrm{E}[X_i \mid A, W_{i-1} = 1]\Pr[W_{i-1} = 1 \mid A] \\ &\le 2^{-k}\Pr[W_{i-1} = 0 \mid A] + 0 \\ &\le 2^{-k}.\end{aligned}$$

So we have $p_i \le 2^{-k}$ for all $1 \le i \le n-k+1$. This gives $\mu = 2^{-k}(n-k+1)$, and $\Pr[X \ge (1+\delta)\mu] \le \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$.

If we want a two-sided bound, we can set $\delta = 1$ (since X can't drop below 0 anyway, and get $\Pr\left[|X - \mathrm{E}[X]| > 2^{-k}(n-k+1)\right] \le \left(\frac{e}{4}\right)^{2^{-k}(n-k+1)}$. This is exponentially small for $k = o(\lg n)$. If k is much bigger than $\lg n$, then we have $\mathrm{E}[X] \ll 1$, so Markov's inequality alone gives us a strong concentration bound.

However, in both cases, the bounds are competitive with the previous bounds from McDiarmid's inequality only if $\mathrm{E}[X] = O(\sqrt{n\log(1/\epsilon)})$. So McDiarmid's inequality wins for $k = o(\log n)$, Markov's inequality wins for $k = \omega(\log n)$, and Chernoff bounds may be useful for a small interval in the middle.

C.3 Assignment 3: due Wednesday, 2011-02-23, at 17:00

C.3.1 Longest common subsequence

A **common subsequence** of two sequences v and w is a sequence u of length k such that there exist indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ with $u_\ell = v_{i_\ell} = w_{j_\ell}$ for all ℓ. For example, `ardab` is a common subsequence of `abracadabra` and `cardtable`.

Let v and w be words of length n over an alphabet of size n drawn independently and uniformly at random. Give the best upper bound you can on the expected length of the longest common subsequence of v and w.

Solution

Let's count the expectation of the number X_k of common subsequences of length k. We have $\binom{n}{k}$ choices of positions in v, and $\binom{n}{k}$ choices of positions in w; for each such choices, there is a probability of exactly n^{-k} that the

corresponding positions match. This gives

$$\begin{aligned}
\mathrm{E}\left[X_k\right] &= \binom{n}{k}^2 n^{-k} \\
&< \frac{n^k}{(k!)^2}. \\
&< \frac{n^k}{(k/e)^{2k}} \\
&= \left(\frac{ne^2}{k^2}\right)^k.
\end{aligned}$$

We'd like this bound to be substantially less than 1. We can't reasonably expect this to happen unless the base of the exponent is less than 1, so we need $k > e\sqrt{n}$.

If $k = (1+\epsilon)e\sqrt{n}$ for any $\epsilon > 0$, then $\mathrm{E}\left[X_k\right] < (1+\epsilon)^{-2e\sqrt{n}} < \frac{1}{n}$ for sufficiently large n. It follows that the expected length of the longest common subsequent is at most $(1+\epsilon)e\sqrt{n}$ for sufficiently large n (because if there are no length-k subsequences, the longest subsequence has length at most $k-1$, and if there is at least one, the longest has length at most n; this gives a bound of at most $(1-1/n)(k-1)+(1/n)n < k$). So in general we have the length of the longest common subsequence is at most $(1+o(1))e\sqrt{n}$.

Though it is not required by the problem, here is a quick argument that the expected length of the longest common subsequence is $\Omega(\sqrt{n})$, based on the **Erdős-Szekeres theorem** [ES35].[1] The Erdős-Szekeres theorem says that any permutation of n^2+1 elements contains either an increasing sequence of $n+1$ elements or a decreasing sequence of $n+1$ elements. Given two random sequences of length n, let S be the set of all elements that appear in both, and consider two permutations ρ and σ of S corresponding to the order in which the elements appear in v and w, respectively (if an element appears multiple times, pick one of the occurrences at random). Then the Erdős-Szekeres theorem says that ρ contains a sequence of length at least $\left\lfloor\sqrt{|\rho|-1}\right\rfloor$ that is either increasing or decreasing with respect to the order given by σ; by symmetry, the probability that it is increasing is at least $1/2$. This gives an expected value for the longest common subsequence that is at least $\mathrm{E}\left[\sqrt{|\rho|-1}\right]/2$.

[1] As suggested by Benjamin Kunsberg.

Let $X = |\rho|$. We can compute a lower bound $\mathrm{E}[X]$ easily; each possible element fails to occur in v with probability $(1-1/n)^n \leq e^{-1}$, and similarly for w. So the chance that an element appears in both sequences is at least $(1-e^{-1})^2$, and thus $\mathrm{E}[X] \geq n(1-e^{-1})^2$. What we want is $\mathrm{E}\left[\sqrt{X-1}\right]/2$; but here the fact that $\sqrt{x}$ is concave means that $\mathrm{E}\left[\sqrt{X-1}\right] \geq \sqrt{\mathrm{E}[X-1]}$ by Jensen's inequality (4.3.1). So we have $\mathrm{E}\left[\sqrt{|\rho|-1}\right]/2 \geq \sqrt{n(1-e^{-1})^2-1}/2 \approx \frac{1-e^{-1}}{2}\sqrt{n}$.

This is not a very good bound (empirically, the real bound seems to be in the neighborhood of $1.9\sqrt{n}$ when $n = 10000$), but it shows that the upper bound of $(1+o(1))e\sqrt{n}$ is tight up to constant factors.

C.3.2 A strange error-correcting code

Let Σ be an alphabet of size $m+1$ that includes m non-blank symbols and a special blank symbol. Let S be a set of $\binom{n}{k}$ strings of length n with non-blank symbols in exactly k positions each, such that no two strings in S have non-blank symbols in the same k positions.

For what value of m can you show S exists such that no two strings in S have the same non-blank symbols in $k-1$ positions?

Solution

This is a job for the Lovász Local Lemma. And it's even symmetric, so we can use the symmetric version (Corollary 11.3.2).

Suppose we assign the non-blank symbols to each string uniformly and independently at random. For each $A \subseteq S$ with $|A| = k$, let X_A be the string that has non-blank symbols in all positions in A. For each pair of subsets A, B with $|A| = |B| = k$ and $|A \cap B| = k-1$, let $C_{A,B}$ be the event that X_A and X_B are identical on all positions in $A \cap B$. Then $\Pr[C_{A,B}] = m^{-k+1}$.

We now now need to figure out how many events are in each neighborhood $\Gamma(C_{A,B})$. Since $C_{A,B}$ depends only on the choices of values for A and B, it is independent of any events $C_{A',B'}$ where neither of A' or B' is equal to A or B. So we can make $\Gamma(C_{A,B})$ consist of all events $C_{A,B'}$ and $C_{A',B}$ where $B' \neq B$ and $A' \neq A$.

For each fixed A, there are exactly $(n-k)k$ events B that overlap it in $k-1$ places, because we can specify B by choosing the elements in $B \setminus A$ and $A \setminus B$. This gives $(n-k)k-1$ events $C_{A,B'}$ where $B' \neq B$. Applying the same argument for A' gives a total of $d = 2(n-k)k-2$ events in $\Gamma(C_{A,B})$. Corollary 11.3.2 applies if $ep(d+1) \leq 1$, which in this case means

$em^{-(k-1)}(2(n-k)k-1) \leq 1$. Solving for m gives

$$m \geq (2e(n-k)k-1)^{1/(k-1)}. \tag{C.3.1}$$

For $k \ll n$, the $(n-k)^{1/(k-1)} \approx n^{1/(k-1)}$ term dominates the shape of the right-hand side asymptotically as k gets large, since everything else goes to 1. This suggests we need $k = \Omega(\log n)$ to get m down to a constant.

Note that (C.3.1) doesn't work very well when $k = 1$.[2] For the $k = 1$ case, there is no overlap in non-blank positions between different strings, so $m = 1$ is enough.

C.3.3 A multiway cut

Given a graph $G = (V, E)$, a **3-way cut** is a set of edges whose endpoints lie in different parts of a partition of the vertices V into three disjoint parts $S \cup T \cup U = V$.

1. Show that any graph with m edges has a 3-way cut with at least $2m/3$ edges.

2. Give an efficient deterministic algorithm for finding such a cut.

Solution

1. Assign each vertex independently to S, T, or U with probability $1/3$ each. Then the probability that any edge uv is contained in the cut is exactly $2/3$. Summing over all edges gives an expected $2m/3$ edges.

2. We'll derandomize the random vertex assignment using the method of conditional probabilities. Given a partial assignment of the vertices, we can compute the conditional expectation of the size of the cut assuming all other vertices are assigned randomly: each edge with matching assigned endpoints contributes 0 to the total, each edge with non-matching assigned endpoints contributes 1, and each edge with zero or one assigned endpoints contributes $2/3$. We'll pick values for the vertices in some arbitrary order to maximize this conditional expectation (since our goal is to get a large cut). At each step, we need only consider the effect on edges incident to the vertex we are assigning whose other endpoints are already assigned, because the contribution

[2] Thanks to Brad Hayes for pointing this out.

of any other edge is not changed by the assignment. Then maximizing the conditional probability is done by choosing an assignment that matches the assignment of the fewest previously-assigned neighbors: in other words, the natural greedy algorithm works. The cost of this algorithm is $O(n+m)$, since we loop over all vertices and have to check each edge at most once for each of its endpoints.

C.4 Assignment 4: due Wednesday, 2011-03-23, at 17:00

C.4.1 Sometimes successful betting strategies are possible

You enter a casino with $X_0 = a$ dollars, and leave if you reach 0 dollars or b or more dollars, where $a, b \in \mathbb{N}$. The casino is unusual in that it offers arbitrary fair games subject to the requirements that:

- Any payoff resulting from a bet must be a nonzero integer in the range $-X_t$ to X_t, inclusive, where X_t is your current wealth.
- The expected payoff must be exactly 0. (In other words, your assets X_t should form a martingale sequence.)

For example, if you have 2 dollars, you may make a bet that pays off -2 with probability $2/5$, $+1$ with probability $2/5$ and $+2$ with probability $1/5$; but you may not make a bet that pays off -3, $+3/2$, or $+4$ under any circumstances, or a bet that pays off -1 with probability $2/3$ and $+1$ with probability $1/3$.

1. What strategy should you use to maximize your chances of leaving with at least b dollars?
2. What strategy should you use to maximize your changes of leaving with nothing?
3. What strategy should you use to maximize the number of bets you make before leaving?

Solution

1. Let X_t be your wealth at time t, and let τ be the stopping time when you leave. Because $\{X_t\}$ is a martingale, $\mathrm{E}[X_0] = a = \mathrm{E}[X_\tau] = \Pr[X_\tau \geq b]\,\mathrm{E}[X_\tau \mid X_\tau \geq b]$. So $\Pr[X_\tau \geq b]$ is maximized by making

$\mathrm{E}[X_\tau \mid X_\tau \geq b]$ as small as possible. It can't be any smaller than b, which can be obtained exactly by making only ± 1 bets. The gives a probability of leaving with b of exactly a/b.

2. Here our goal is to minimize $\Pr[X_\tau \geq b]$, so we want to make $\mathrm{E}[X_\tau \mid X_\tau \geq b]$ as large as possible. The largest value of X_τ we can possibly reach is $2(b-1)$; we can obtain this value by betting ± 1 until we reach $b-1$, then making any fair bet with positive payoff $b-1$ (for example, $\pm(b-1)$ with equal probability works, as does a bet that pays off $b-1$ with probability $1/b$ and -1 with probability $(b-1)/b$). In this case we get a probability of leaving with 0 of $1 - \frac{a}{2(b-1)}$.

3. For each t, let $\delta_t = X_t - X_{t-1}$ and $V_t = \mathrm{Var}[\delta_t \mid \mathcal{F}_t]$. We have previously shown (see the footnote to §8.4.1) that $\mathrm{E}\left[X_\tau^2\right] = \mathrm{E}\left[X_0^2\right] + \mathrm{E}\left[\sum_{t=1}^{\tau} V_t\right]$ where τ is an appropriate stopping time. When we stop, we know that $X_\tau^2 \leq (2(b-1))^2$, which puts an upper bound on $\mathrm{E}\left[\sum_{i=1}^{\tau} V_i\right]$. We can spend this bound most parsimoniously by minimizing V_i as much as possible. If we make each $\delta_t = \pm 1$, we get the smallest possible value for V_t (since any change contributes at least 1 to the variance). However, in this case we don't get all the way out to $2(b-1)$ at the high end; instead, we stop at b, giving an expected number of steps equal to $a(b-a)$.

   We can do a bit better than this by changing our strategy at $b-1$. Instead of betting ± 1, let's pick some x and place a bet that pays off $b-1$ with probability $\frac{1}{b}$ and -1 with probability $\frac{b-1}{b} = 1 - \frac{1}{b}$. (The idea here is to minimize the conditional variance while still allowing ourselves to reach $2(b-1)$.) Each ordinary random walk step has $V_t = 1$; a "big" bet starting at $b-1$ has $V_t = 1 - \frac{1}{b} + \frac{(b-1)^2}{b} = \frac{b-1+b^2-2b+1}{b} = b - \frac{1}{b}$.

   To analyze this process, observe that starting from a, we first spending $a(b-a-1)$ steps on average to reach either 0 (with probability $1 - \frac{a}{b-1}$ or $b-1$ (with probability $\frac{a}{b-1}$. In the first case, we are done. Otherwise, we take one more step, then with probability $\frac{1}{b}$ we lose and with probability $\frac{b-1}{b}$ we continue starting from $b-2$. We can write a recurrence for our expected number of steps $T(a)$ starting from a, as:

$$T(a) = a(b-a-1) + \frac{a}{b-1}\left(1 + \frac{b-1}{b}T(b-2)\right). \qquad \text{(C.4.1)}$$

When $a = b - 2$, we get

$$\begin{aligned} T(b-2) &= (b-2) + \frac{b-2}{b-1}\left(1 + \frac{b-1}{b}T(b-2)\right) \\ &= (b-2)\left(1 + \frac{1}{b-1}\right) + \frac{b-2}{b}T(b-2), \end{aligned}$$

which gives

$$\begin{aligned} T(b-2) &= \frac{(b-2)\frac{2b-1}{b-1}}{2/b} \\ &= \frac{b(b-2)(2b-1)}{2(b-1)}. \end{aligned}$$

Plugging this back into (C.4.1) gives

$$\begin{aligned} T(a) &= a(b-a-1) + \frac{a}{b-1}\left(1 + \frac{b-1}{b}\frac{b(b-2)(2b-1)}{2(b-1)}\right) \\ &= ab - a^2 + a + \frac{a}{b-1} + \frac{a(b-2)(2b-1)}{2(b-1)} \\ &= \frac{3}{2}ab + O(b). \end{aligned} \tag{C.4.2}$$

This is much better than the $a(b-a)$ value for the straight ± 1 strategy, especially when a is also large.

I don't know if this particular strategy is in fact optimal, but that's what I'd be tempted to bet.

C.4.2 Random walk with reset

Consider a random walk on $\mathbb{N}$ that goes up with probability 1/2, down with probability 3/8, and resets to 0 with probability 1/8. When $X_t > 0$, this gives:

$$X_{t+1} = \begin{cases} X_t + 1 & \text{with probability } 1/2, \\ X_t - 1 & \text{with probability } 3/8, \text{ and} \\ 0 & \text{with probability } 1/8. \end{cases}$$

When $X_t = 0$, we let $X_{t+1} = 1$ with probability 1/2 and 0 with probability 1/2.

1. What is the stationary distribution of this process?

2. What is the mean recurrence time μ_n for some state n?

3. Use μ_n to get a tight asymptotic (i.e., big-Θ) bound on $\mu_{0,n}$, the expected time to reach n starting from 0.

Solution

1. For $n > 0$, we have $\pi_n = \frac{1}{2}\pi_{n-1} + \frac{3}{8}\pi_{n+1}$, with a base case $\pi_0 = \frac{1}{8} + \frac{3}{8}\pi_0 + \frac{3}{8}\pi_1$.

   The π_n expression is a linear homogeneous recurrence, so its solution consists of linear combinations of terms b^n, where b satisfies $1 = \frac{1}{2}b^{-1} + \frac{3}{8}b$. The solutions to this equation are $b = 2/3$ and $b = 2$; we can exclude the $b = 2$ case because it would make our probabilities blow up for large n. So we can reasonably guess $\pi_n = a(2/3)^n$ when $n > 0$.

   For $n = 0$, substitute $\pi_0 = \frac{1}{8} + \frac{3}{8}\pi_0 + \frac{3}{8}a(2/3)$ to get $\pi_0 = \frac{1}{5} + \frac{2}{5}a$. Now substitute

$$\begin{aligned}
\pi_1 &= (2/3)a \\
&= \frac{1}{2}\pi_0 + \frac{3}{8}a(2/3)^2 \\
&= \frac{1}{2}\left(\frac{1}{5} + \frac{2}{5}a\right) + \frac{3}{8}a(2/3)^2 \\
&= \frac{1}{10} + \frac{11}{30}a,
\end{aligned}$$

   which we can solve to get $a = 1/3$.

   So our candidate π is $\pi_0 = 1/3$, $\pi_n = (1/3)(2/3)^n$, and in fact we can drop the special case for π_0.

   As a check, $\sum_{i=0}^{n} \pi_n = (1/3)\sum_{i=0}^{n}(2/3)^n = \frac{1/3}{1-2/3} = 1$.

2. Since $\mu_n = 1/\pi_n$, we have $\mu_n = 3(3/2)^n$.

3. In general, let $\mu_{k,n}$ be the expected time to reach n starting at k. Then $\mu_n = \mu_{n,n} = 1 + \frac{1}{8}\mu_{0,n} + \frac{1}{2}\mu_{n+1,n} + \frac{3}{8}\mu_{n-1,n} \geq 1 + \mu_{0,n}/8$. It follows that $\mu_{0,n} \leq 8\mu_n + 1 = 24(3/2)^n + 1 = O((3/2)^n)$.

   For the lower bound, observe that $\mu_n \leq \mu_{n,0} + \mu_{0,n}$. Since there is a 1/8 chance of reaching 0 from any state, we have $\mu_{n,0} \leq 8$. It follows that $\mu_n \leq 8 + \mu_{0,n}$ or $\mu_{0,n} \geq \mu_n - 8 = \Omega((3/2)^n)$.

C.4.3 Yet another shuffling algorithm

Suppose we attempt to shuffle a deck of n cards by picking a card uniformly at random, and swapping it with the top card. Give the best bound you can on the mixing time for this process to reach a total variation distance of ϵ from the uniform distribution.

Solution

It's tempting to use the same coupling as for move-to-top (see §9.4.5). This would be that at each step we choose the same card to swap to the top position, which increases by at least one the number of cards that are in the same position in both decks. The problem is that at the next step, these two cards are most likely separated again, by being swapped with other cards in two different positions.

Instead, we will do something slightly more clever. Let Z_t be the number of cards in the same position at time t. If the top cards of both decks are equal, we swap both to the same position chosen uniformly at random. This has no effect on Z_t. If the top cards of both decks are not equal, we pick a card uniformly at random and swap it to the top in both decks. This increases Z_t by at least 1, unless we happen to pick cards that are already in the same position; so Z_t increases by at least 1 with probability $1 - Z_t/n$.

Let's summarize a state by an ordered pair (k, b) where $k = Z_t$ and b is 0 if the top cards are equal and 1 if they are not equal. Then we have a Markov chain where $(k, 0)$ goes to $(k, 1)$ with probability $\frac{n-k}{n}$ (and otherwise stays put); and $(k, 1)$ goes to $(k+1, 0)$ (or higher) with probability $\frac{n-k}{n}$ and to $(k, 0)$ with probability $\frac{k}{n}$.

Starting from $(k, 0)$, we expect to wait $\frac{n}{n-k}$ steps on average to reach $(k, 1)$, at which point we move to $(k + 1, 0)$ or back to $(k, 0)$ in one more step; we iterate through this process $\frac{n}{n-k}$ times on average before we are successful. This gives an expected number of steps to get from $(k, 0)$ to $(k + 1, 0)$ (or possibly a higher value) of $\frac{n}{n-k}\left(\frac{n}{n-k} + 1)\right)$. Summing over k up to $n - 2$ (since once $k > n - 2$, we will in fact have $k = n$, since k can't

be $n-1$), we get

$$\begin{aligned}
\mathrm{E}[\tau] &\leq \sum_{k=0}^{n-2} \frac{n}{n-k}\left(\frac{n}{n-k}+1\right) \\
&= \sum_{m=2}^{n} \left(\frac{n^2}{m^2}+\frac{n}{m}\right) \\
&\leq n^2\left(\frac{\pi^2}{6}-1\right)+n\ln n. \\
&= O(n^2).
\end{aligned}$$

So we expect the deck to mix in $O(n^2 \log(1/\epsilon))$ steps. (I don't know if this is the real bound; my guess is that it should be closer to $O(n \log n)$ as in all the other shuffling procedures.)

C.5 Assignment 5: due Thursday, 2011-04-07, at 23:59

C.5.1 A reversible chain

Consider a random walk on $\mathbb{Z}_m$, where $p_{i,i+1} = 2/3$ for all i and $p_{i,i-1} = 1/3$ for all i except $i = 0$. Is it possible to assign values to $p_{0,m-1}$ and $p_{0,0}$ to make this chain reversible, and if so, what stationary distribution do you get?

Solution

Suppose we can make this chain reversible, and let π be the resulting stationary distribution. From the detailed balance equations, we have $(2/3)\pi_i = (1/3)\pi_{i+1}$ or $\pi_{i+1} = 2\pi_i$ for $i = 0 \ldots m-2$. The solution to this recurrence is $\pi_i = 2^i \pi_0$, which gives $\pi_i = \frac{2^i}{2^m-1}$ when we set π_0 to get $\sum_i \pi_i = 1$.

Now solve $\pi_0 p_{0,m-1} = \pi_{m-1} p_{m-1,0}$ to get

$$\begin{aligned}
p_{0,m-1} &= \frac{\pi_{m-1} p_{m-1,0}}{\pi_0} \\
&= 2^{m-1}(2/3) \\
&= 2^m/3.
\end{aligned}$$

This is greater than 1 for $m > 1$, so except for the degenerate cases of $m = 1$ and $m = 2$, it's not possible to make the chain reversible.

C.5.2 Toggling bits

Consider the following Markov chain on an array of n bits $a[1], a[2], \ldots a[n]$. At each step, we choose a position i uniformly at random. We then change $A[i]$ to $\neg A[i]$ with probability $1/2$, provided $i = 1$ or $A[i-1] = 1$ (if neither condition holds hold, do nothing).[3]

1. What is the stationary distribution?
2. How quickly does it converge?

Solution

1. First let's show irreducibility. Starting from an arbitrary configuration, repeatedly switch the leftmost 0 to a 1 (this is always permitted by the transition rules); after at most n steps, we reach the all-1 configuration. Since we can repeat this process in reverse to get to any other configuration, we get that every configuration is reachable from every other configuration in at most $2n$ steps ($2n-1$ if we are careful about handling the all-0 configuration separately).

   We also have that for any two adjacent configurations x and y, $p_{xy} = p_{yx} = \frac{1}{2n}$. So we have a reversible, irreducible, aperiodic (because there exists at least one self-loop) chain with a uniform stationary distribution $\pi_x = 2^{-n}$.

2. Here is a bound using the obvious coupling, where we choose the same position in X and Y and attempt to set it to the same value. To show this coalesces, given X_t and Y_t define Z_t to be the position of the rightmost 1 in the common prefix of X_t and Y_t, or 0 if there is no 1 in the common prefix of X_t and Y_t. Then Z_t increases by at least 1 if we attempt to set position $Z_t + 1$ to 1, which occurs with probability $\frac{1}{2n}$, and decreases by at most 1 if we attempt to set Z_t to 0, again with probability $\frac{1}{2n}$.

   It follows that Z_t reaches n no later than a ± 1 random walk on $0 \ldots n$ with reflecting barriers that takes a step every $1/n$ time units on average. The expected number of steps to reach n from the worst-case

[3] Motivation: Imagine each bit represents whether a node in some distributed system is inactive (0) or active (1), and you can only change your state if you have an active left neighbor to notify. Also imagine that there is an always-active *base station* at -1 (alternatively, imagine that this assumption makes the problem easier than the other natural arrangement where we put all the nodes in a ring).

starting position of 0 is exactly n^2. (Proof: model the random walk with a reflecting barrier at 0 by folding a random walk with absorbing barriers at $\pm n$ in half, then use the bound from §8.4.1.) We must then multiply this by n to get an expected n^3 steps in the original process. So the two copies coalesce in at most n^3 expected steps. My suspicion is one could improve this bound with a better analysis by using the bias toward increasing Z_t to get the expected time to coalesce down to $O(n^2)$, but I don't know any clean way to do this.

The path coupling version of this is that we look at two adjacent configurations X_t and Y_t, use the obvious coupling again, and see what happens to $\mathrm{E}\left[d(X_{t+1}, Y_{t+1}) \mid X_t, Y_t\right]$, where the distance is the number of transitions needed to convert X_t to Y_t or vice versa. If we pick the position i where X_t and Y_t differ, then we coalesce; this occurs with probability $1/n$. If we change the 1 to the left of i to a 0, then $d(X_{t+1}, Y_{t+1})$ rises to 3 (because to get from X_{t+1} to Y_{t+1}, we have to change position $i-1$ to 1, change position i, and then change position $i-1$ back to 0); this occurs with probability $1/2n$ if $i > 1$. But we can also get into trouble if we try to change position $i+1$; we can only make the change in one of X_t and Y_t, so we get $d(X_{t+1}, Y_{t+1}) = 2$ in this case, which occurs with probability $1/2n$ when $i < n$. Adding up all three cases gives a worst-case expected change of $-1/n + 2/2n + 1/2n = 1/2n > 0$. So unless we can do something more clever, path coupling won't help us here.

However, it is possible to get a bound using canonical paths, but the best bound I could get was not as good as the coupling bound. The basic idea is that we will change x into y one bit at a time (from left to right, say), so that we will go through a sequence of intermediate states of the form $y[1]y[2]\ldots y[i]x[i+1]x[i+2]\ldots x[n]$. But to change $x[i+1]$ to $y[i+1]$, we may also need to reach out with a tentacle of 1 bits from from the last 1 in the current prefix of y (and then retract it afterwards). Given a particular transition where we change a 0 to a 1, we can reconstruct the original x and y by specifying (a) which bit i at or after our current position we are trying to change; (b) which 1 bit before our current position is the last "real" 1 bit in y as opposed to something we are creating to reach out to position i; and (c) the values of $x[1]\ldots x[i-1]$ and $y[i+1]\ldots y[i]$. A similar argument applies to $1 \to 0$ transitions. So we are routing at most $n^2 2^{n-1}$ paths across

each transition, giving a bound on the congestion

$$\begin{aligned}\rho &\leq \left(\frac{1}{2^{-n}/2n}\right) n^2 2^{n-1} 2^{-2n} \\ &= n^3.\end{aligned}$$

The bound on τ_2 that follows from this is $8n^6$, which is pretty bad (although the constant could be improved by counting the (a) and (b) bits more carefully). As with the coupling argument, it may be that there is a less congested set of canonical paths that gives a better bound.

C.5.3 Spanning trees

Suppose you have a connected graph $G = (V, E)$ with n nodes and m edges. Consider the following Markov process. Each state H_t is a subgraph of G that is either a spanning tree or a spanning tree plus an additional edge. At each step, flip a fair coin. If it comes up heads, choose an edge e uniformly at random from E and let $H_{t+1} = H_t \cup \{e\}$ if H_t is a spanning tree and let $H_{t+1} = H_t \setminus \{e\}$ if H_t is not a spanning tree and $H_t \setminus \{e\}$ is connected. If it comes up tails and H_t is a spanning tree, let H_{t+1} be some other spanning tree, sampled uniformly at random. In all other cases, let $H_{t+1} = H_t$.

Let N be the number of states in this Markov chain.

1. What is the stationary distribution?
2. How quickly does it converge?

Solution

1. Since every transition has a matching reverse transition with the same transition probability, the chain is reversible with a uniform stationary distribution $\pi_H = 1/N$.
2. Here's a coupling that coalesces in at most $4m/3 + 2$ expected steps:
   (a) If X_t and Y_t are both trees, then send them to the same tree with probability 1/2; else let them both add edges independently (or we could have them add the same edge—it doesn't make any difference to the final result).
   (b) If only one of X_t and Y_t is a tree, with probability 1/2 scramble the tree while attempting to remove an edge from the non-tree,

and the rest of the time scramble the non-tree (which has no effect) while attempting to add an edge to the tree. Since the non-tree has at least three edges that can be removed, this puts $(X_{t+1}, Y_{t+1}$ in the two-tree case with probability at least $3/2m$.

(c) If neither X_t nor Y_t is a tree, attempt to remove an edge from both. Let S and T be the sets of edges that we can remove from X_t and Y_t, respectively, and let $k = \min(|S|, |T|) \geq 3$. Choose k edges from each of S and T and match them, so that if we remove one edge from each pair, we also remove the other edge. As in the previous case, this puts $(X_{t+1}, Y_{t+1}$ in the two-tree case with probability at least $3/2m$.

To show this coalesces, starting from an arbitrary state, we reach a two-tree state in at most $2m/3$ expected steps. After one more step, we either coalesce (with probability $1/2$) or restart from a new arbitrary state. This gives an expected coupling time of at most $2(2m/3+1) = 4m/3+2$ as claimed.

C.6 Assignment 6: due Monday, 2011-04-25, at 17:00

C.6.1 Sparse satisfying assignments to DNFs

Given a formula in disjunctive normal form, we'd like to estimate the number of satisfying assignments in which exactly w of the variables are true. Give a fully polynomial-time randomized approximation scheme for this problem.

Solution

Essentially, we're going to do the Karp-Luby covering trick [KL85] described in §10.3, but will tweak the probability distribution when we generate our samples so that we only get samples with weight w.

Let U be the set of assignment with weight w (there are exactly $\binom{n}{w}$ such assignments, where n is the number of variables). For each clause C_i, let $U_i = \{x \in U \mid C_i(x) = 1\}$. Now observe that:

1. We can compute $|U_i|$. Let $k_i = |C_i|$ be the number of variables in C_i and $k_i^+ = \left|C_i^+\right|$ the number of variables that appear in positive form in C_i. Then $|U_i| = \binom{n-k_i}{w-k_i^+}$ is the number of ways to make a total of w variables true using the remaining $n - k_i$ variables.

2. We can sample uniformly from U_i, by sampling a set of $w - k_i^+$ true variables not in C_i uniformly from all variables not in C_i.

3. We can use the values computed for $|U_i|$ to sample i proportionally to the size of $|U_i|$.

So now we sample pairs (i, x) with $x \in U_i$ uniformly at random by sampling i first, then sampling $x \in U_i$. As in the original algorithm, we then count (i, x) if and only if C_i is the leftmost clause for which $C_i(x) = 1$. The same argument that at least $1/m$ of the (i, x) pairs count applies, and so we get the same bounds as in the original algorithm.

C.6.2 Detecting duplicates

Algorithm C.1 attempts to detect duplicate values in an input array S of length n.

```
Initialize A[1...n] to ⊥
Choose a hash function h
for i ← 1...n do
    x ← S[i]
    if A[h(x)] = x then
        return true
    else
        A[h(x)] ← x
return false
```

Algorithm C.1: Dubious duplicate detector

It's easy to see that Algorithm C.1 never returns **true** unless some value appears twice in S. But maybe it misses some duplicates it should find.

1. Suppose h is a random function. What is the worst-case probability that Algorithm C.1 returns **false** if S contains two copies of some value?

2. Is this worst-case probability affected if h is drawn instead from a 2-universal family of hash functions?

Solution

1. Suppose that $S[i] = S[j] = x$ for $i < j$. Then the algorithm will see x in $A[h(x)]$ on iteration j and return **true**, unless it is overwritten by some value $S[k]$ with $i < k < j$. This occurs if $h(S[k]) = h(x)$, which occurs with probability exactly $1-(1-1/n)^{j-i-1}$ if we consider all possible k. This quantity is maximized at $1-(1-1/n)^{n-2} \approx 1-(1-1/n)^2/e \approx 1-(1-1/2n)/e$ when $i = 1$ and $j = n$.

2. As it happens, the algorithm can fail pretty badly if all we know is that h is 2-universal. What we can show is that the probability that some $S[k]$ with $i < j < k$ gets hashed to the same place as $x = S[i] = S[j]$ in the analysis above is at most $(j-i-1)/n \leq (n-2)/n = (1-2/n)$, since each $S[k]$ has at most a $1/n$ chance of colliding with x and the union bound applies. But it is possible to construct a 2-universal family for which we get exactly this probability in the worst case.

   Let $U = \{0 \ldots n\}$, and define for each a in $\{0 \ldots n-1\}$ $h_a : U \to n$ by $h_a(n) = 0$ and $h_a(x) = (x + a) \bmod n$ for $x \neq n$. Then $H = \{h_a\}$ is 2-universal, since if $x \neq y$ and neither x nor y is n, $\Pr[h_a(x) = h_a(y)] = 0$, and if one of x or y is n, $\Pr[h_a(x) = h_a(y)] = 1/n$. But if we use this family in Algorithm C.1 with $S[1] = S[n] = n$ and $S[k] = k$ for $1 < k < n$, then there are $n-2$ choices of a that put one of the middle values in $A[0]$.

C.6.3 Balanced Bloom filters

A clever algorithmist decides to solve the problem of Bloom filters filling up with ones by capping the number of ones at $m/2$. As in a standard Bloom filter, an element is inserted by writing ones to $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$; but after writing each one, if the number of one bits in the bit-vector is more than $m/2$, one of the ones in the vector (chosen uniformly at random) is changed back to a zero.

Because some of the ones associated with a particular element might be deleted, the membership test answers yes if at least 3/4 of the bits $A[h_1(x)] \ldots A[h_k(x)]$ are ones.

To simplify the analysis, you may assume that the h_i are independent random functions. You may also assume that $(3/4)k$ is an integer.

1. Give an upper bound on the probability of a false positive when testing for a value x that has never been inserted.

2. Suppose that we insert x at some point, and then follow this insertion with a sequence of insertions of new, distinct values $y_1, y_2, \ldots$. Assuming a worst-case state before inserting x, give asymptotic upper and lower bounds on the expected number of insertions until a test for x fails.

Solution

1. The probability of a false positive is maximized when exactly half the bits in A are one. If x has never been inserted, each $A[h_i(x)]$ is equally likely to be zero or one. So $\Pr[\text{false positive for } x] = \Pr[S_k \geq (3/4)k]$ when S_k is a binomial random variable with parameters $1/2$ and k. Chernoff bounds give

$$\begin{aligned}\Pr[S_k \geq (3/4)k] &= \Pr[S_k \geq (3/2)\,\mathrm{E}[S_k]] \\ &\leq \left(\frac{e^{1/2}}{(3/2)^{3/2}}\right)^{k/2} \\ &\leq (0.94734)^k.\end{aligned}$$

   We can make this less than any fixed ϵ by setting $k \geq 20\ln(1/\epsilon)$ or thereabouts.

2. For false negatives, we need to look at how quickly the bits for x are eroded away. A minor complication is that the erosion may start even as we are setting $A[h_1(x)] \ldots A[h_k(x)]$.

   Let's consider a single bit $A[i]$ and look at how it changes after (a) setting $A[i] = 1$, and (b) setting some random $A[r] = 1$.

   In the first case, $A[i]$ will be 1 after the assignment unless it is set back to zero, which occurs with probability $\frac{1}{m/2+1}$. This distribution does not depend on the prior value of $A[i]$.

   In the second case, if $A[i]$ was previously 0, it becomes 1 with probability

$$\begin{aligned}\frac{1}{m}\left(1 - \frac{1}{m/2+1}\right) &= \frac{1}{m} \cdot \frac{m/2}{m/2+1} \\ &= \frac{1}{m+2}.\end{aligned}$$

If it was previously 1, it becomes 0 with probability

$$\frac{1}{2} \cdot \frac{1}{m/2+1} = \frac{1}{m+2}.$$

So after the initial assignment, $A[i]$ just flips its value with probability $\frac{1}{m+2}$.

It is convenient to represent $A[i]$ as ± 1; let $X_i^t = -1$ if $A[i] = 0$ at time t, and 1 otherwise. Then X_i^t satisfies the recurrence

$$\begin{aligned} \mathrm{E}\left[X_i^{t+1} \;\middle|\; X_i^t\right] &= \frac{m+1}{m+2} X_i^t - \frac{1}{m+2} X_i^t \\ &= \frac{m}{m+2} X_i^t. \\ &= \left(1 - \frac{2}{m+2}\right) X_i^t. \end{aligned}$$

We can extend this to $\mathrm{E}\left[X_i^t \mid X_i^0\right] = \left(1 - \frac{2}{m+2}\right)^t X_i^0 \approx e^{-2t/(m+2)} X_i^0$.

Similarly, after setting $A[i] = 1$, we get $\mathrm{E}\left[X_i\right] = 1 - 2\frac{1}{m/2+1} = 1 - \frac{4}{2m+1} = 1 - o(1)$.

Let $S^t = \sum_{i=1}^k X_{h_i(x)}^t$. Let 0 be the time at which we finish inserting x. Then each for each i we have

$$1 - o(1)e^{-2k/(m+2)} \leq \mathrm{E}\left[X_{h_i(x)}^0\right] \leq 1 - o(1),$$

from which it follows that

$$k(1 - o(1))e^{-2k/(m+2)} \leq \mathrm{E}\left[S^0\right] \leq 1 - o(1)$$

and in general that

$$k(1 - o(1))e^{-2(k+t)/(m+2)} \leq \mathrm{E}\left[S^t\right] \leq 1 - o(1)e^{-2t/(m+2)}.$$

So for any fixed $0 < \epsilon < 1$ and sufficiently large m, we will have $\mathrm{E}\left[S^t\right] = \epsilon k$ for some t' where $t \leq t' \leq k + t$ and $t = \Theta(m \ln(1/\epsilon))$.

We are now looking for the time at which S^t drops below $k/2$ (the $k/2$ is because we are working with ± 1 variables). We will bound when this time occurs using Markov's inequality.

Let's look for the largest time t with $\mathrm{E}\left[S^t\right] \geq (3/4)k$. Then $\mathrm{E}\left[k - S^t\right] \leq k/4$ and $\Pr\left[k - S^t \geq k/2\right] \leq 1/2$. It follows that after $\Theta(m) - k$ operations, x is still visible with probability $1/2$, which implies that the expected time at which it stops being visible is at least $(\Omega(m) - k)/2$. To get the expected number of insert operations, we divide by k, to get $\Omega(m/k)$.

For the upper bound, apply the same reasoning to the first time at which $\mathrm{E}\left[S^t\right] \leq k/4$. This occurs at time $O(m)$ at the latest (with a different constant), so after $O(m)$ steps there is at most a $1/2$ probability that $S^t \geq k/2$. If S^t is still greater than $k/2$ at this point, try again using the same analysis; this gives us the usual geometric series argument that $\mathrm{E}\left[t\right] = O(m)$. Again, we have to divide by k to get the number of insert operations, so we get $O(m/k)$ in this case.

Combining these bounds, we have that x disappears after $\Theta(m/k)$ insertions on average. This seems like about what we would expect.

C.7 Final exam

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

C.7.1 Leader election

Suppose we have n processes and we want to elect a leader. At each round, each process flips a coin, and drops out if the coin comes up tails. We win if in some round there is exactly one process left.

Let $T(n)$ be the probability that this event eventually occurs starting with n processes. For small n, we have $T(0) = 0$ and $T(1) = 1$. Show that there is a constant $c > 0$ such that $T(n) \geq c$ for all $n > 1$.

Solution

Let's suppose that there is some such c. We will necessarily have $c \leq 1 = T(1)$, so the induction hypothesis will hold in the base case $n = 1$.

For $n \geq 2$, compute

$$\begin{aligned} T(n) &= \sum_{k=0}^{n} 2^{-n} \binom{n}{k} T(k) \\ &= 2^{-n}T(n) + 2^{-n}nT(1) + \sum_{k=2}^{n-1} 2^{-n} \binom{n}{k} T(k) \\ &\geq 2^{-n}T(n) + 2^{-n}n + 2^{-n}(2^n - n - 2)c. \end{aligned}$$

Solve for $T(n)$ to get

$$\begin{aligned} T(n) &\geq \frac{n + (2^n - n - 2)c}{2^n - 1} \\ &= c\left(\frac{2^n - n - 2 + n/c}{2^n - 1}\right). \end{aligned}$$

This will be greater than or equal to c if $2^n - n - 2 + n/c \geq 2^n - 1$ or $n/c \geq n+1$, which holds if $c \leq \frac{n}{n+1}$. The worst case is $n = 2$ giving $c = 2/3$.

Valiant and Vazirani [VV86] used this approach to reduce solving general instances of SAT to solving instances of SAT with unique solutions; they prove essentially the result given above (which shows that fixing variables in a SAT formula is likely to produce a SAT formula with a unique solution at some point) with a slightly worse constant.

C.7.2 Two-coloring an even cycle

Here is a not-very-efficient algorithm for 2-coloring an even cycle. Every node starts out red or blue. At each step, we pick one of the n nodes uniformly at random, and change its color if it has the same color as at least one of its neighbors. We continue until no node has the same color as either of its neighbors.

Suppose that in the initial state there are exactly two monochromatic edges. What is the worst-case expected number of steps until there are no monochromatic edges?

Solution

Suppose we have a monochromatic edge surrounded by non-monochrome edges, e.g. $RBRRBR$. If we pick one of the endpoints of the edge (say the left endpoint in this case), then the monochromatic edge shifts in the direction of that endpoint: $RBBRBRB$. Picking any node not incident to a

monochromatic edge has no effect, so in this case there is no way to increase the number of monochromatic edges.

It may also be that we have two adjacent monochromatic edges: $BRRRB$. Now if we happen to pick the node in the middle, we end up with no monochromatic edges ($BRBRB$) and the process terminates. If on the other hand we pick one of the nodes on the outside, then the monochromatic edges move away from each other.

We can thus model the process with 2 monochromatic edges as a random walk, where the difference between the leftmost nodes of the edges (mod n) increases or decreases with equal probability $2/n$ except if the distance is 1 or -1; in this last case, the distance increases (going to 2 or -2) with probability $2/n$, but decreases only with probability $1/n$. We want to know when this process hits 0 (or n).

Imagine a random walk starting from position k with absorbing barriers at 1 and $n-1$. This reaches 1 or $n-1$ after $(k-1)(n-1-k)$ steps on average, which translates into $(n/4)(k-1)(n-1-k)$ steps of our original process if we take into account that we only move with probability $4/n$ per time unit. This time is maximized by setting $k = n/2$, which gives $(n/4)(n/2-1)^2 = n^3/16 - n^2/4 + n/4$ expected time units to reach 1 or $n-1$ for the first time.

At 1 or $n-1$, we wait an addition $n/3$ steps on average; then with probability $1/3$ the process finishes and with probability $2/3$ we start over from position 2 or $n-2$; in the latter case, we run $(n/4)(n-3)+n/3$ time units on average before we may finish again. On average, it takes 3 attempts to finish. Each attempt incurs the expected $n/3$ cost before taking a step, and all but the last attempt incur the expected $(n/4)(n-3)$ additional steps for the random walk. So the last phase of the process the process adds $(1/2)n(n-3)+n = (1/2)n^2 - (5/4)n$ steps.

Adding up all of the costs gives $n^3/16 - n^2/4 + n/4 + n/3 + (1/2)n^2 - (5/4)n = \frac{1}{16}n^3 + \frac{1}{4}n^2 - \frac{2}{3}n$ steps.

C.7.3 Finding the maximum

Suppose that we run Algorithm C.2 on an array with n elements, all of which are distinct. What is the expected number of times Line 5 is executed as a function of n?

```
1 Randomly permute A
2 m ← −∞
3 for i ← 1 . . . n do
4     if A[i] > m then
5         m ← A[i]
6 return m
```

Algorithm C.2: Randomized max-finding algorithm

Solution

Let X_i be the indicator variable for the event that Line 5 is executed on the i-th pass through the loop. This will occur if $A[i]$ is greater than $A[j]$ for all $j < i$, which occurs with probability exactly $1/i$ (given that A has been permuted randomly). So the expected number of calls to Line 5 is $\sum i = 1^n \mathrm{E}[X_i] = \sum_{i=1}^n \frac{1}{i} = H_n$.

C.7.4 Random graph coloring

Let G be a random d-regular graph on n vertices, that is, a graph drawn uniformly from the family of all n-vertex graph in which each vertex has exactly d neighbors. Color the vertices of G red or blue independently at random.

1. What is the expected number of monochromatic edges in G?
2. Show that the actual number of monochromatic edges is tightly concentrated around its expectation.

Solution

The fact that G is itself a random graph is a red herring; all we really need to know is that it's d-regular.

1. Because G has exactly $dn/2$ edges, and each edge has probability $1/2$ of being monochromatic, the expected number of monochromatic edges is $dn/4$.
2. This is a job for Azuma's inequality. Consider the vertex exposure martingale. Changing the color of any one vertex changes the number of monochromatic edges by at most d. So we have $\Pr[|X - \mathrm{E}[X]| \geq t] \leq$

$2\exp\left(-t^2/2\sum_{i=1}^{n} d^2\right) = 2e^{-t^2/2nd^2}$, which tells us that the deviation is likely to be not much more than $O(d\sqrt{n})$.

Appendix D

Sample assignments from Spring 2009

D.1 Final exam, Spring 2009

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

D.1.1 Randomized mergesort (20 points)

Consider the following randomized version of the mergesort algorithm. We take an unsorted list of n elements and split it into two lists by flipping an independent fair coin for each element to decide which list to put it in. We then recursively sort the two lists, and merge the resulting sorted lists. The merge procedure involves repeatedly comparing the smallest element in each of the two lists and removing the smaller element found, until one of the lists is empty.

Compute the expected number of comparisons needed to perform this final merge. (You do not need to consider the cost of performing the recursive sorts.)

Solution

Color the elements in the final merged list red or blue based on which sublist they came from. The only elements that do not require a comparison to insert into the main list are those that are followed only by elements of the

same color; the expected number of such elements is equal to the expected length of the longest monochromatic suffix. By symmetry, this is the same as the expected longest monochromatic prefix, which is equal to the expected length of the longest sequence of identical coin-flips.

The probability of getting k identical coin-flips in a row followed by a different coin-flip is exactly 2^{-k}; the first coin-flip sets the color, the next $k-1$ must follow it (giving a factor of 2^{-k+1}, and the last must be the opposite color (giving an additional factor of 2^{-1}). For n identical coin-flips, there is a probability of 2^{-n+1}, since we don't need an extra coin-flip of the opposite color. So the expected length is $\sum_{k=1}^{n-1} k2^{-k} + n2^{-n+1} = \sum_{k=0}^{n} k2^{-k} + n2^{-n}$.

We can simplify the sum using generating functions. The sum $\sum_{k=0}^{n} 2^{-k} z^k$ is given by $\frac{1-(z/2)^{n+1}}{1-z/2}$. Taking the derivative with respect to z gives $\sum_{k=0}^{n} 2^{-k} k z^{k-1} = (1/2)\frac{1-(z/2)^{n+1}}{1-z/2}^2 + (1/2)\frac{(n+1)(z/2)^n}{1-z/2}$. At $z = 1$ this is $2(1 - 2^{-n-1}) - 2(n + 1)2^{-n} = 2 - (n+2)2^{-n}$. Adding the second term gives $\mathrm{E}[X] = 2 - 2 \cdot 2^{-n} = 2 - 2^{-n+1}$.

Note that this counts the expected number of elements for which we do not have to do a comparison; with n elements total, this leaves $n - 2 + 2^{-n+1}$ comparisons on average.

D.1.2 A search problem (20 points)

Suppose you are searching a space by generating new instances of some problem from old ones. Each instance is either good or bad; if you generate a new instance from a good instance, the new instance is also good, and if you generate a new instance from a bad instance, the new instance is also bad.

Suppose that your start with X_0 good instances and Y_0 bad instances, and that at each step you choose one of the instances you already have uniformly at random to generate a new instance. What is the expected number of good instances you have after n steps?

Hint: Consider the sequence of values $\{X_t/(X_t + Y_t)\}$.

Solution

We can show that the suggested sequence is a martingale, by computing

$$\begin{aligned}
\mathrm{E}\left[\frac{X_{t+1}}{X_{t+1}+Y_{t+1}} \;\middle|\; X_t, Y_t\right] &= \frac{X_t}{X_t+Y_t}\cdot\frac{X_t+1}{X_t+Y_t+1} + \frac{Y_t}{X_t+Y_t}\cdot\frac{X_t}{X_t+Y_t+1} \\
&= \frac{X_t(X_t+1)+Y_tX_t}{(X_t+Y_t)(X_t+Y_t+1)} \\
&= \frac{X_t(X_t+Y_t+1)}{(X_t+Y_t)(X_t+Y_t+1)} \\
&= \frac{X_t}{X_t+Y_t}.
\end{aligned}$$

From the martingale property we have $\mathrm{E}\left[\frac{X_n}{X_n+Y_n}\right] = \frac{X_0}{X_0+Y_0}$. But $X_n + Y_n = X_0 + Y_0 + n$, a constant, so we can multiply both sides by this value to get $\mathrm{E}[X_n] = X_0\left(\frac{X_0+Y_0+n}{X_0+Y_0}\right)$.

D.1.3 Support your local police (20 points)

At one point I lived in a city whose local police department supported themselves in part by collecting fines for speeding tickets. A speeding ticket would cost 1 unit (approximately \$100), and it was unpredictable how often one would get a speeding ticket. For a price of 2 units, it was possible to purchase a metal placard to go on your vehicle identifying yourself as a supporter of the police union, which (at least according to local legend) would eliminate any fines for subsequent speeding tickets, but which would not eliminate the cost of any previous speeding tickets.

Let us consider the question of when to purchase a placard as a problem in on-line algorithms. It is possible to achieve a strict[1] competitive ratio of 2 by purchasing a placard after the second ticket. If one receives fewer than 2 tickets, both the on-line and off-line algorithms pay the same amount, and at 2 or more tickets the on-line algorithm pays 4 while the off-line pays 2 (the off-line algorithm purchased the placard before receiving any tickets at all).

1. Show that no deterministic algorithm can achieve a lower (strict) competitive ratio.

2. Show that a randomized algorithm can do so, against an oblivious adversary.

[1] I.e., with no additive constant.

Solution

1. Any deterministic algorithm essentially just chooses some fixed number m of tickets to collect before buying the placard. Let n be the actual number of tickets issued. For $m = 0$, the competitive ratio is infinite when $n = 0$. For $m = 1$, the competitive ratio is 3 when $n = 1$. For $m > 2$, the competitive ratio is $(m + 2)/2 > 2$ when $n = m$. So $m = 2$ is the optimal choice.

2. Consider the following algorithm: with probability p, we purchase a placard after 1 ticket, and with probability $q = 1 - p$, we purchase a placard after 2 tickets. This gives a competitive ratio of 1 for $n = 0$, $1 + 2p$ for $n = 1$, and $(3p + 4q)/2 = (4 - p)/2 = 2 - p/2$ for $n \geq 2$. There is a clearly a trade-off between the two ratios $1+2p$ and $2-p/2$. The break-even point is when they are equal, at $p = 2/5$. This gives a competitive ratio of $1 + 2p = 9/5$, which is less than 2.

D.1.4 Overloaded machines (20 points)

Suppose n^2 jobs are assigned to n machines with each job choosing a machine independently and uniformly at random. Let the load on a machine be the number of jobs assigned to it. Show that for any fixed $\delta > 0$ and sufficiently large n, there is a constant $c < 1$ such that the maximum load exceeds $(1 + \delta)n$ with probability at most nc^n.

Solution

This is a job for Chernoff bounds. For any particular machine, the load S is a sum of independent indicator variables and the mean load is $\mu = n$. So we have

$$\Pr\left[S \geq (1+\delta)\mu\right] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^n.$$

Observe that $e^\delta/(1+\delta)^{1+\delta} < 1$ for $\delta > 0$. One proof of this fact is to take the log to get $\delta-(1+\delta)\log(1+\delta)$, which equals 0 at $\delta = 0$, and then show that the logarithm is decreasing by showing that $\frac{d}{d\delta}\cdots = 1 - \frac{1+\delta}{1+\delta} - \log(1+\delta) = -\log(1+\delta) < 0$ for all $\delta > 0$.

So we can let $c = e^\delta/(1+\delta)^{1+\delta}$ to get a bound of c^n on the probability that any particular machine is overloaded and a bound of nc^n (from the union bound) on the probability that any of the machines is overloaded.

Appendix E

Probabilistic recurrences

Randomized algorithms often produce recurrences with ugly sums embedded inside them (see, for example, (1.3.1)). We'd like to have tools for pulling at least asymptotic bounds out of these recurrences without having to deal with the sums. This isn't always possible, but for certain simple recurrences we can make it work.

E.1 Recurrences with constant cost functions

Let us consider probabilistic recurrences of the form $T(n) = 1 + T(n - X_n)$, where X_n is a random variable with $0 < X_n \leq n$ and $T(0) = 0$. We assume we can compute a lower bound on $\mathrm{E}[X_n]$ for each n, and we want to translate this lower bound into an upper bound on $\mathrm{E}[T(n)]$.

E.2 Examples

- How long does it take to get our first heads if we repeatedly flip a coin that comes up heads with probability p? Even though we probably already know the answer to this, We can solve it by solving the recurrence $T(1) = 1 + T(1 - X_1), T(0) = 0$, where $\mathrm{E}[X_1] = p$.

- **Hoare's FIND** [Hoa61b], often called **QuickSelect**, is an algorithm for finding the k-th smallest element of an unsorted array. It works like QuickSort, only after partitioning the array around a random pivot we throw away the part that doesn't contain our target and recurse only on the surviving piece. How many rounds of this must we do? Here $\mathrm{E}[X_n]$ is more complicated, since after splitting our array of size n

into piles of size n' and $n - n' - 1$, we have to pick one or the other (or possibly just the pivot alone) based on the value of k.

- Suppose we start with n biased coins that each come up heads with probability p. In each round, we flip all the coins and throw away the ones that come up tails. How many rounds does it take to get rid of all of the coins? (This essentially tells us how tall a skip list [Pug90] can get.) Here we have $\mathrm{E}[X_n] = (1-p)n$.

- In the **coupon collector problem**, we sample from $1 \ldots n$ with replacement until we see ever value at least once. We can model this by a recurrence in which $T(k)$ is the time to get all the coupons given there are k left that we haven't seen. Here X_n is 1 with probability k/n and 0 with probability $(n-k)/n$, giving $\mathrm{E}[X_n] = k/n$.

- Let's play **Chutes and Ladders** without the chutes and ladders. We start at location n, and whenever it's our turn, we roll a fair six-sided die X and move to $n - X$ unless this value is negative, in which case we stay put until the next turn. How many turns does it take to get to 0?

E.3 The Karp-Upfal-Wigderson bound

This is a bound on the expected number of rounds to finish a process where we start with a problem instance of size n, and after one round of work we get a new problem instance of size $n - X_n$, where X_n is a random variable whose distribution depends on n. It was original described in a paper by Karp, Upfal, and Wigderson on analyzing parallel search algorithms [KUW88]. The bound applies when $\mathrm{E}[X_n]$ is bounded below by a non-decreasing function $\mu(n)$.

Lemma E.3.1. *Let a be a constant, let $T(n) = 1 + T(n - X_n)$, where for each n, X_n is an integer-valued random variable satisfying $0 \le X_n \le n - a$ and let $T(a) = 0$. Let $\mathrm{E}[X_n] \ge \mu(n)$ for all $n > a$, where μ is a positive non-decreasing function of n. Then*

$$\mathrm{E}[T(n)] \le \int_a^n \frac{1}{\mu(t)}\, dt. \tag{E.3.1}$$

To get an intuition for why this works, imagine that X_n is the speed at which we drop from n, expressed in units per round. Traveling at this speed, it takes $1/X_n$ rounds to cross from $k+1$ to k for any such interval

we pass. From the point of view of the interval $[k, k+1]$, we don't know which n we are going to start from before we cross it, but we do know that for any $n \geq k+1$ we start from, our speed will be at least $\mu(n) \geq \mu(k+1)$ on average. So the time it takes will be at most $\int_k^{k+1} \frac{1}{\mu(t)}\, dt$ on average, and the total time is obtained by summing all of these intervals.

Of course, this intuition is not even close to a real proof (among other things, there may be a very dangerous confusion in there between $1/\operatorname{E}[X_n]$ and $\operatorname{E}[1/X_n]$), so we will give a real proof as well.

Proof of Lemma E.3.1. This is essentially the same proof as in Motwani and Raghavan [MR95], but we add some extra detail to allow for the possibility that $X_n = 0$.

Let $p = \Pr[X_n = 0]$, $q = 1 - p = \Pr[X_n \neq 0]$. Note we have $q > 0$ because otherwise $\operatorname{E}[X_n] = 0 < \mu(n)$. Then we have

$$\begin{aligned}
\operatorname{E}[T(n)] &= 1 + \operatorname{E}[T(n - X_n)] \\
&= 1 + p \operatorname{E}[T(n - X_n) \mid X_n = 0] + q \operatorname{E}[T(n - X_n) \mid X_n \neq 0] \\
&= 1 + p \operatorname{E}[T(n)] + q \operatorname{E}[T(n - X_n) \mid X_n \neq 0].
\end{aligned}$$

Now we have $\operatorname{E}[T(n)]$ on both sides, which we don't like very much. So we collect it on the left-hand side:

$$(1 - p) \operatorname{E}[T(n)] = 1 + q \operatorname{E}[T(n - X_n) \mid X_n \neq 0],$$

divide both sides by $q = 1 - p$, and apply the induction hypothesis:

$$\begin{aligned}
\operatorname{E}[T(n)] &= 1/q + \operatorname{E}[T(n - X_n) \mid X_n \neq 0] \\
&= 1/q + \operatorname{E}[\operatorname{E}[T(n - X_n) \mid X_n] \mid X_n \neq 0] \\
&\leq 1/q + \operatorname{E}\left[\int_a^{n - X_n} \frac{1}{\mu(t)}\, dt \;\middle|\; X_n \neq 0\right] \\
&= 1/q + \operatorname{E}\left[\int_a^{n} \frac{1}{\mu(t)}\, dt - \int_{n - X_n}^{n} \frac{1}{\mu(t)}\, dt \;\middle|\; X_n \neq 0\right] \\
&\leq 1/q + \int_a^{n} \frac{1}{\mu(t)}\, dt - \operatorname{E}\left[\frac{X_n}{\mu(n)} \;\middle|\; X_n \neq 0\right] \\
&\leq 1/q + \int_a^{n} \frac{1}{\mu(t)}\, dt - \frac{\operatorname{E}[X_n \mid X_n \neq 0]}{\mu(n)}.
\end{aligned}$$

The second-to-last step uses the fact that $\mu(t) \leq \mu(n) for t \leq n$.

It may seem like we don't know what $\operatorname{E}[X_n \mid X_n \neq 0]$ is. But we know that $X_n \geq 0$, so we have $\operatorname{E}[X_n] = p \operatorname{E}[X_n \mid X_n = 0] + q \operatorname{E}[X_n \mid X_n \neq 0] =$

$q\,\mathrm{E}\left[X_n \mid X_n \neq 0\right]$. So we can solve for $\mathrm{E}\left[X_n \mid X_n \neq 0\right] = E[X_n]/q$. So let's plug this in:

$$\begin{aligned}\mathrm{E}\left[T(n)\right] &\leq 1/q + \int_a^n \frac{1}{\mu(t)}\,dt - \frac{\mathrm{E}\left[X_n\right]/q}{\mu(n)} \\ &\leq 1/q + \int_a^n \frac{1}{\mu(t)}\,dt - 1/q \\ &= \int_a^n \frac{1}{\mu(t)}\,dt.\end{aligned}$$

This concludes the proof. □

Now we just need to find some applications.

E.3.1 Waiting for heads

For the recurrence $T(1) = 1+T(1-X_1)$ with $\mathrm{E}\left[X_1\right] = p$, we set $\mu(n) = p$ and get $\mathrm{E}\left[T(1)\right] \leq \int_0^1 \frac{1}{p}\,dt = \frac{1}{p}$, which happens to be exactly the right answer.

E.3.2 Quickselect

In Quickselect, we pick a random pivot and split the original array of size n into three piles of size m (less than the pivot), 1 (the pivot itself), and $n-m-1$ (greater than the pivot). We then figure out which of the three piles contains the k-th smallest element (depend on how k compares to $m-1$) and recurse, stopping when we hit a pile with 1 element. It's easiest to analyze this by assuming that we recurse in the largest of the three piles, i.e., that our recurrence is $T(n) = 1 + \max(T(m), T(n-m-1))$, where m is uniform in $0 \ldots n-1$. The exact value of $\mathrm{E}\left[\max(m, n-m-1)\right]$ is a little messy to compute (among other things, it depends on whether n is odd or even), but it's not hard to see that it's always less than $(3/4)n$. So letting $\mu(n) = n/4$, we get

$$\mathrm{E}\left[T(n)\right] \leq \int_1^n \frac{1}{t/4}\,dt = 4\ln n.$$

E.3.3 Tossing coins

Here we have $\mathrm{E}\left[X_n\right] = (1-p)n$. If we let $\mu(n) = (1-p)n$ and plug into the formula without thinking about it too much, we get

$$\mathrm{E}\left[T(n)\right] \leq \int_0^n \frac{1}{(1-p)t}\,dt = \frac{1}{1-p}(\ln n - \ln 0).$$

That $\ln 0$ is trouble. We can fix it by making $\mu(n) = (1-p)\lceil n \rceil$, to get

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \int_{0+}^{n} \frac{1}{(1-p)\lceil t \rceil}\, dt \\
&= \frac{1}{1-p} \sum_{k=1}^{n} \frac{1}{k} \\
&= \frac{H_n}{1-p}.
\end{aligned}$$

E.3.4 Coupon collector

Now that we know how to avoid dividing by zero, this is easy and fun. Let $\mu(x) = \lceil x \rceil / n$, then we have

$$\begin{aligned}
\mathrm{E}[T(n)] &\le \int_{0+}^{n} \frac{n}{\lceil t \rceil}\, dt \\
&= \sum_{k=1}^{n} \frac{n}{k} \\
&= nH_n.
\end{aligned}$$

As it happens, this is the exact answer for this case. This will happen whenever X is always a 0–1 variable[1] and we define $\mu(x) = \mathrm{E}[X \mid n = \lceil x \rceil]$, which can be seen by spending far too much time thinking about the precise sources of error in the inequalities in the proof.

E.3.5 Chutes and ladders

Let $\mu(n)$ be the expected drop from position n. We have to be a little bit careful about small n, but we can compute that in general $\mu(n) = \frac{1}{6}\sum_{i=0}^{\min(n,6)} i$. For fractional values x we will set $\mu(x) = \mu(\lceil x \rceil)$ as before.

[1] A **0–1 random variable** is also called a **Bernoulli random variable**, but 0–1 is shorter to type and more informative. Even more confusing, the underlying experiment that gives rise to a Bernoulli random variable goes by the entirely different name of a **Poisson trial**. Jacob Bernoulli and Siméon-Denis Poisson were great mathematicians, but there are probably better ways to remember them.

Then we have

$$\begin{aligned} \mathrm{E}\left[T(n)\right] &\leq \int_{0+}^{n} \frac{1}{\mu(t)}\, dt \\ &= \sum_{k=1}^{n} \frac{1}{\mu(k)} \end{aligned}$$

We can summarize the values in the following table:

n	$\mu(n)$	$1/\mu(n)$	$\sum 1/\mu(k)$
1	1/6	6	6
2	1/2	2	8
3	1	1	9
4	5/3	3/5	48/5
5	5/2	2/5	10
≥ 6	7/2	2/7	$10 + (2/7)(n-5) = (2/7)n + 65/7$

This is a slight overestimate; for example, we can calculate by hand that the expected waiting time for $n = 2$ is 6 and for $n = 3$ that it is $20/3 = 6 + 2/3$.

We can also consider the generalized version of the game where we start at n and drop by $1 \ldots n$ each turn as long as the drop wouldn't take us below 0. Now the expected drop from position k is $k(k+1)/2n$, and so we can apply the formula to get

$$\mathrm{E}\left[T(n)\right] \leq \sum_{k=1}^{n} \frac{2n}{k(k+1)}.$$

The sum of $\frac{1}{k(k+1)}$ when k goes from 1 to n happens to have a very nice value; it's exactly $\frac{n}{n+1} = 1 + \frac{1}{n+1}$.[2] So in this case we can rewrite the bound as $2n \cdot \frac{n}{n+1} = \frac{2n^2}{n+1}$.

E.4 High-probability bounds

So far we have only considered bounds on the expected value of $T(n)$. Suppose we want to show that $T(n)$ is in fact small with high probability, i.e.,

[2]Proof: Trivially true for $n = 0$; for larger n, compute $\sum_{k=1}^{n} \frac{1}{k(k+1)} \sum_{k=1}^{n-1} \frac{1}{k(k+1)} + \frac{1}{n(n+1)} = \frac{n-1}{n} + \frac{1}{n(n+1)} = \frac{(n-1)(n+1)-1}{n(n+1)} = \frac{n^2}{n(n+1)} = n/(n+1)$.

a statement of the form $\Pr[T(n) \geq t] \leq \epsilon$. There are two natural ways to do this: we can repeatedly apply Markov's inequality to the expectation bound, or we can attempt to analyze the recurrence in more detail. The first method tends to give weaker bounds but it's easier.

E.4.1 High-probability bounds from expectation bounds

Given $\mathrm{E}[T(n)] \leq m$, we have $\Pr[T(n) \geq \alpha m] \leq \alpha^{-1}$. This does not give a very good bound on probability; if we want to show $\Pr[T(n) \geq t] \leq n^{-c}$ for some constant c (a typical high-probability bound), we need $t \geq n^c m$. But we can get a better bound if m bounds the expected time starting from any reachable state, as is the case for the class of problems we have been considering.

The idea is that if $T(n)$ exceeds αm, we restart the analysis and argue that $\Pr[T(n) \geq 2\alpha m \mid T(n) \geq \alpha m] \leq \alpha^{-1}$, from which it follows that $\Pr[T(n) \geq 2\alpha m] \leq \alpha^{-2}$. In general, for any non-negative integer k, we have $\Pr[T(n) \geq k\alpha m] \leq \alpha^{-k}$. Now we just need to figure out how to pick α to minimize this quantity for fixed t.

Let $t = k\alpha m$. Then $k = t/\alpha m$ and we seek to minimize $\alpha^{-t/\alpha m}$. Taking the logarithm gives $-(t/m)(\ln \alpha)/\alpha$. The t/m factor is irrelevant to the minimization problem, so we are left with minimizing $-(\ln \alpha)/\alpha$. Taking the derivative gives $-\alpha^{-2} + \alpha^{-2} \ln \alpha$; this is zero when $\ln \alpha = 1$ or $\alpha = e$. (This popular constant shows up often in problems like this.) So we get $\Pr[T(n) \geq kem] \leq e^{-k}$, or, letting $k = \ln(1/\epsilon)$, $\Pr[T(n) \geq em \ln(1/\epsilon)] \leq \epsilon$.

So, for example, we can get an n^{-c} bound on the probability of running too long by setting our time bound to $em \ln(n^c) = cem \ln n = O(m \log n)$. We can't hope to do better than $O(m)$, so this bound is tight up to a log factor.

E.4.2 Detailed analysis of the recurrence

As Lance Fortnow has explained,[3] getting rid of log factors is what theoretical computer science is all about. So we'd like to do better than an $O(m \log n)$ bound if we can. In some cases this is not too hard.

Suppose for each n, $T(n) = 1 + T(X)$, where $\mathrm{E}[X] \leq \alpha n$ for a fixed constant α. Let $X_0 = n$, and let X_1, X_2, etc., be the sequence of sizes of the remaining problem at time 1, 2, etc. Then we have $\mathrm{E}[X_1] \leq \alpha n$ from our assumption. But we also have $\mathrm{E}[X_2] = \mathrm{E}[\mathrm{E}[X_2 \mid X_1]] \leq \mathrm{E}[\alpha X_1] = \alpha \mathrm{E}[X_1] \leq \alpha^2 n$, and by induction we can show that $\mathrm{E}[X_k] \leq \alpha^k n$ for all

[3] `http://weblog.fortnow.com/2009/01/soda-and-me.html`

k. Since X_k is integer-valued, $\mathrm{E}[X_k]$ is an upper bound on $\Pr[X_k > 0]$; we thus get $\Pr[T(n) \geq k] = \Pr[X_k > 0] \leq \alpha^k n$. We can solve for the value of k that makes this less than ϵ: $k = -\log(n/\epsilon)/\log\alpha = \log_{1/\alpha} n + \log_{1/\alpha}(1/\epsilon)$.

For comparison, the bound on the expectation of $T(n)$ from Lemma E.3.1 is $H(n)/(1-\alpha)$. This is actually pretty close to $\log_{1/\alpha} n$ when α is close to 1, and is not too bad even for smaller α. But the difference is that the dependence on $\log(1/\epsilon)$ is additive with the tighter analysis, so for fixed c we get $\Pr[T(n) \geq t] \leq n^{-c}$ at $t = O(\log n + \log n^c) = O(\log n)$ instead of $O(\log n \log n^c) = O(\log^2 n)$.

E.5 More general recurrences

We didn't do these, but if we had to, it might be worth looking at Roura's Improved Master Theorems [Rou01].

Bibliography

[AA11] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, September 2011.

[ABMRT96] Arne Andersson, Peter Bro Miltersen, Søren Riis, and Mikkel Thorup. Static dictionaries on AC^0 RAMs: Query time $\theta(\sqrt{\log n/\log\log n})$ is necessary and sufficient. In *FOCS*, pages 441–450, 1996.

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002.

[Ach03] Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, June 2003.

[Adl78] Leonard Adleman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 75–83, Washington, DC, USA, 1978. IEEE Computer Society.

[AF01] David Aldous and James Allen Fill. Reversible Markov chains and random walks on graphs. Unpublished manuscript, available at `http://www.stat.berkeley.edu/~aldous/RWG/book.html`, 2001.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[Alo91] Noga Alon. A parallel algorithmic version of the local lemma. *Random Structures & Algorithms*, 2(4):367–378, 1991.

[AS92] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley & Sons, 1992.

[AS07] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, November 2007.

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An information organization algorithm. *Doklady Akademia Nauk SSSR*, 146:263–266, 1962.

[AW96] James Aspnes and Orli Waarts. Randomized consensus in $o(n \log^2 n)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[Azu67] Kazuoki Azuma. Weighted sums of certain dependent random variables. *Tôhoku Mathematical Journal*, 19(3):357–367, 1967.

[Bab79] László Babai. Monte-carlo algorithms in graph isomorphism testing. Technical Report D.M.S. 79-10, Université de Montréal, 1979.

[BBBV97] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, October 1997.

[BCB12] Sébastian Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 5(1):1–122, 2012.

[BD92] Dave Bayer and Persi Diaconis. Trailing the dovetail shuffle to its lair. *Annals of Applied Probability*, 2(2):294–313, 1992.

[Bec91] József Beck. An algorithmic approach to the lovász local lemma. i. *Random Structures & Algorithms*, 2(4):343–365, 1991.

[Bel57] Richard Earnest Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[Bol01] Béla Bollobás. *Random Graphs*. Cambridge University Press, second edition, 2001.

[Bro86] Andrei Z. Broder. How hard is to marry at random? (on the approximation of the permanent). In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing, 28-30 May 1986, Berkeley, California, USA*, pages 50–58, 1986.

[Bro88] Andrei Z. Broder. Errata to "how hard is to marry at random? (on the approximation of the permanent)". In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, 2-4 May 1988, Chicago, Illinois, USA*, page 551, 1988.

[BRSW06] Boaz Barak, Anup Rao, Ronen Shaltiel, and Avi Wigderson. 2-source dispersers for sub-polynomial entropy and Ramsey graphs beating the Frankl-Wilson construction. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, STOC '06, pages 671–680, New York, NY, USA, 2006. ACM.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CM03] Saar Cohen and Yossi Matias. Spectral bloom filters. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 241–252, 2003.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CS00] Artur Czumaj and Christian Scheideler. Coloring non-uniform hypergraphs: a new algorithmic approach to the general Lovász local lemma. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 30–39,

Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[CW77] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM.

[Deu89] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989.

[Dev88] Luc Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26(1-2):123–130, October 1988.

[DG00] Martin Dyer and Catherine Greenhill. On Markov chains for independent sets. *J. Algorithms*, 35:17–49, April 2000.

[DGM02] Martin Dyer, Catherine Greenhill, and Mike Molloy. Very rapid mixing of the Glauber dynamics for proper colorings on bounded-degree graphs. *Random Struct. Algorithms*, 20:98–114, January 2002.

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

[Dir39] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35:416–418, 6 1939.

[DJ92] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[Dum56] A. I. Dumey. Indexing for rapid random-access memory. *Computers and Automation*, 5(12):6–9, 1956.

[Dye03] Martin Dyer. Approximate counting by dynamic programming. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 693–699, New York, NY, USA, 2003. ACM.

[EL75] Paul Erdős and Laszlo Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal, R. Rado, and V. T. Sós, editors, *Infinite and Finite Sets (to Paul Erdős on his 60th birthday)*, pages 609–627. North-Holland, 1975.

[Erd45] P. Erdős. On a lemma of Littlewood and Offord. *Bulletin of the American Mathematical Society*, 51(12):898–902, 1945.

[Erd47] P. Erdős. Some remarks on the theory of graphs. *Bulletin of the American Mathematical Society*, 53:292–294, 1947.

[ES35] P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

[Eul68] M. L. Euler. Remarques sur un beau rapport entre les séries des pussances tant directes que réciproques. *Mémoires de l'Académie des Sciences de Berlin*, 17:83–106, 1768.

[FCAB00] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, third edition, 1968.

[Fel71] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. Wiley, second edition, 1971.

[FGQ12] Xiequan Fan, Ion Grama, and Liu Quansheng. Hoeffding's inequality for supermartingales. Available as `http://arxiv.org/abs/1109.4359`, July 2012.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a Sparse Table with O(1) Worst Case Access Time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[FNM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In Helmut Alt and Michel Habib, editors, *STACS*

2003, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. Springer Berlin Heidelberg, 2003.

[FR75] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975.

[GKP88] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1988.

[Goo83] Nelson Goodman. *Fact, Fiction, and Forecast*. Harvard University Press, 1983.

[GR93] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1993.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219. ACM, 1996. Available as `http://arxiv.org/abs/quant-ph/9605043`.

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978.

[GS92] G. R. Grimmet and D. R. Stirzaker. *Probability and Random Processes*. Oxford Science Publications, 2nd edition, 1992.

[GS01] Geoffrey R. Grimmett and David R. Stirzaker. *Probability and Random Processes*. Oxford University Press, 2001.

[Gur00] Venkatesen Guruswami. Rapidly mixing Markov chains: A comparison of techniques. Available at `ftp://theory.lcs.mit.edu/pub/people/venkat/markov-survey.ps`, 2000.

[GW94] Michel X. Goemans and David P. Williamson. New 3/4-approximation algorithms for the maximum satisfiability problem. *SIAM J. Discret. Math.*, 7:656–666, November 1994.

[GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[GW12] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012.

[Has70] W. K. Hastings. Monte carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[HH80] P. Hall and C.C. Heyde. *Martingale Limit Theory and Its Application*. Academic Press, 1980.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[Hoa61a] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4:321, July 1961.

[Hoa61b] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4:321–322, July 1961.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.

[HST08] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[JL84] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in Modern Analysis and Probability (New Haven, Connecticut,*

1982), number 26 in Contemporary Mathematics, pages 189–206. American Mathematical Society, 1984.

[JLR00] Svante Janson, Tomasz Łuczak, and Andrezej Ruciński. *Random Graphs*. John Wiley & Sons, 2000.

[JS89] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, 1989.

[JSV04] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.

[Kar93] David R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 21–30, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[KL85] Richard M. Karp and Michael Luby. Monte-carlo algorithms for the planar multiterminal network reliability problem. *J. Complexity*, 1(1):45–64, 1985.

[KM08] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 41–52, New York, NY, USA, 2010. ACM.

[Kol33] A.N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, 1933.

[KR99] V. S. Anil Kumar and H. Ramesh. Markovian coupling vs. conductance for the Jerrum-Sinclair chain. In *FOCS*, pages 241–252, 1999.

[KS76] J.G. Kemeny and J.L. Snell. *Finite Markov Chains: With a New Appendix "Generalization of a Fundamental Matrix"*. Undergraduate Texts in Mathematics. Springer, 1976.

[KSK76] John G. Kemeny, J. Laurie. Snell, and Anthony W. Knapp. *Denumerable Markov Chains*, volume 40 of *Graduate Texts in Mathematics*. Springer, 1976.

[KT75] Samuel Karlin and Howard M. Taylor. *A First Course in Stochastic Processes*. Academic Press, second edition, 1975.

[KUW88] Richard M. Karp, Eli Upfal, and Avi Wigderson. The complexity of parallel search. *Journal of Computer and System Sciences*, 36(2):225–253, 1988.

[Li80] Shuo-Yen Robert Li. A martingale approach to the study of occurrence of sequence patterns in repeated experiments. *Annals of Probability*, 8(6):1171–1176, 1980.

[Lin92] Torgny Lindvall. *Lectures on the Coupling Method*. Wiley, 1992.

[LR85] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, March 1985.

[Lub85] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM.

[LV97] Michael Luby and Eric Vigoda. Approximately counting up to four (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 682–687, New York, NY, USA, 1997. ACM.

[LV99] Michael Luby and Eric Vigoda. Fast convergence of the Glauber dynamics for sampling independent sets. *Random Structures & Algorithms*, 15(3–4):229–241, 1999.

[LW05] Michael Luby and Avi Wigderson. Pairwise independence and derandomization. *Foundations and Trends in Theoretical Computer Science*, 1(4):237–301, 2005.

[McC85] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[McD89] Colin McDiarmid. On the method of bounded differences. In *Surveys in Combinatorics, 1989: Invited Papers at the Twelfth British Combinatorial Conference*, pages 148–188, 1989.

[MH92] Colin McDiarmid and Ryan Hayward. Strong concentration for Quicksort. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete algorithms*, SODA '92, pages 414–421, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 149–159. Morgan Kaufmann, 1986.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[Mos09] Robin A. Moser. A constructive proof of the Lovász local lemma. In *Proceedings of the 41st annual ACM Symposium on Theory of Computing*, STOC '09, pages 343–350, New York, NY, USA, 2009. ACM.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MR98] Michael Molloy and Bruce Reed. Further algorithmic aspects of the local lemma. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 524–529, New York, NY, USA, 1998. ACM.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.

[MT10] Robin A. Moser and Gábor Tardos. A constructive proof of the general Lovász local lemma. *J. ACM*, 57:11:1–11:15, February 2010.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[MWW07] Elchanan Mossel, Dror Weitz, and Nicholas Wormald. On the hardness of sampling independent sets beyond the tree threshold. Available as `http://arxiv.org/abs/math/0701471`, 2007.

[Pag01] Rasmus Pagh. On the cell probe complexity of membership and perfect hashing. In *STOC*, pages 425–432, 2001.

[Pag06] Rasmus Pagh. Cuckoo hashing for undergraduates. Available at `http://www.it-c.dk/people/pagh/papers/cuckoo-undergrad.pdf`, 2006.

[Pap91] Christos H. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 163–169. IEEE Computer Society, 1991.

[PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 823–829. SIAM, 2005.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[PT12] Mihai Patrascu and Mikkel Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):14, 2012.

[Pug90] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[Rab80] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[Rag88] Prabhakar Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37:130–143, October 1988.

[Rou01] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *J. ACM*, 48:170–205, March 2001.

[RT87] Prabhakar Raghavan and Clark D. Tompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, December 1987.

[SA96] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. Available at `http://people.ischool.berkeley.edu/~aragon/pubs/rst96.pdf`.

[She11] Irina Shevtsova. On the asymptotically exact constants in the Berry-Esseen-Katz inequality. *Theory of Probability & Its Applications*, 55(2):225–252, 2011. A preprint is available at `http://arxiv.org/pdf/1111.6554.pdf`.

[Sho97] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.

[Spu12] Francis Spufford. *Red Plenty*. Graywolf Press, 2012.

[Sri08] Aravind Srinivasan. Improved algorithmic versions of the Lovász local lemma. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 611–620, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3–4):281–292, 1971.

[SS87] E. Shamir and J. Spencer. Sharp concentration of the chromatic number on random graphs $G_{n,p}$. *Combinatorica*, 7:121–129, January 1987.

[Str03] Gilbert Strang. *Introduction to linear algebra*. Wellesley-Cambridge Press, 2003.

[Tod91] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991.

[Tof80] Tommaso Toffoli. Reversible computing. Technical Report MIT/LCS/TM-151, MIT Laboratory for Computer Science, 1980.

[Val79] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[Val82] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM J. Comput.*, 11(2):350–361, 1982.

[VB81] Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *STOC*, pages 263–277. ACM, 1981.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

[VV86] Leslie G. Valiant and Vijay V. Vazirani. Np is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.

[Wil04] David Bruce Wilson. Mixing times of lozenge tiling and card shuffling Markov chains. *Annals of Applied Probability*, 14(1):274–325, 2004.

[Wil06] Herbert S. Wilf. *generatingfunctionology*. A. K. Peters, third edition, 2006. Second edition is available on-line at `http://www.math.penn.edu/~wilf/DownldGF.html`.

Index