Chapter 13

Randomized Algorithms

The idea that a process can be "random" is not a modern one; we can trace the notion far back into the history of human thought and certainly see its reflections in gambling and the insurance business, each of which reach into ancient times. Yet, while similarly intuitive subjects like geometry and logic have been treated mathematically for several thousand years, the mathematical study of probability is surprisingly young; the first known attempts to seriously formalize it came about in the 1600s. Of course, the history of computer science plays out on a much shorter time scale, and the idea of randomization has been with it since its early days.

Randomization and probabilistic analysis are themes that cut across many areas of computer science, including algorithm design, and when one thinks about random processes in the context of computation, it is usually in one of two distinct ways. One view is to consider the world as behaving randomly: One can consider traditional algorithms that confront randomly generated input. This approach is often termed *average-case analysis*, since we are studying the behavior of an algorithm on an "average" input (subject to some underlying random process), rather than a worst-case input.

A second view is to consider algorithms that behave randomly: The world provides the same worst-case input as always, but we allow our algorithm to make random decisions as it processes the input. Thus the role of randomization in this approach is purely internal to the algorithm and does not require new assumptions about the nature of the input. It is this notion of a *randomized algorithm* that we will be considering in this chapter.

Why might it be useful to design an algorithm that is allowed to make random decisions? A first answer would be to observe that by allowing randomization, we've made our underlying model more powerful. Efficient deterministic algorithms that always yield the correct answer are a special case of efficient randomized algorithms that only need to yield the correct answer with high probability; they are also a special case of randomized algorithms that are always correct, and run efficiently *in expectation*. Even in a worst-case world, an algorithm that does its own "internal" randomization may be able to offset certain worst-case phenomena. So problems that may not have been solvable by efficient deterministic algorithms may still be amenable to randomized algorithms.

But this is not the whole story, and in fact we'll be looking at randomized algorithms for a number of problems where there exist comparably efficient deterministic algorithms. Even in such situations, a randomized approach often exhibits considerable power for further reasons: It may be conceptually much simpler; or it may allow the algorithm to function while maintaining very little internal state or memory of the past. The advantages of randomization seem to increase further as one considers larger computer systems and networks, with many loosely interacting processes—in other words, a *distributed system*. Here random behavior on the part of individual processes can reduce the amount of explicit communication or synchronization that is required; it is often valuable as a tool for *symmetry-breaking* among processes, reducing the danger of contention and "hot spots." A number of our examples will come from settings like this: regulating access to a shared resource, balancing load on multiple processors, or routing packets through a network. Even a small level of comfort with randomized heuristics can give one considerable leverage in thinking about large systems.

A natural worry in approaching the topic of randomized algorithms is that it requires an extensive knowledge of probability. Of course, it's always better to know more rather than less, and some algorithms are indeed based on complex probabilistic ideas. But one further goal of this chapter is to illustrate *how little* underlying probability is really needed in order to understand many of the well-known algorithms in this area. We will see that there is a small set of useful probabilistic tools that recur frequently, and this chapter will try to develop the tools alongside the algorithms. Ultimately, facility with these tools is as valuable as an understanding of the specific algorithms themselves.

13.1 A First Application: Contention Resolution

We begin with a first application of randomized algorithms—contention resolution in a distributed system—that illustrates the general style of analysis

we will be using for many of the algorithms that follow. In particular, it is a chance to work through some basic manipulations involving *events* and their probabilities, analyzing intersections of events using *independence* as well as unions of events using a simple *Union Bound*. For the sake of completeness, we give a brief summary of these concepts in the final section of this chapter (Section 13.15).

The Problem

Suppose we have n processes $P_1, P_2, \ldots, P_n$, each competing for access to a single shared database. We imagine time as being divided into discrete *rounds*. The database has the property that it can be accessed by at most one process in a single round; if two or more processes attempt to access it simultaneously, then all processes are "locked out" for the duration of that round. So, while each process wants to access the database as often as possible, it's pointless for all of them to try accessing it in every round; then everyone will be perpetually locked out. What's needed is a way to divide up the rounds among the processes in an equitable fashion, so that all processes get through to the database on a regular basis.

If it is easy for the processes to communicate with one another, then one can imagine all sorts of direct means for resolving the contention. But suppose that the processes can't communicate with one another at all; how then can they work out a protocol under which they manage to "take turns" in accessing the database?

Designing a Randomized Algorithm

Randomization provides a natural protocol for this problem, which we can specify simply as follows. For some number $p > 0$ that we'll determine shortly, each process will attempt to access the database in each round with probability p, independently of the decisions of the other processes. So, if exactly one process decides to make the attempt in a given round, it will succeed; if two or more try, then they will all be locked out; and if none try, then the round is in a sense "wasted." This type of strategy, in which each of a set of identical processes randomizes its behavior, is the core of the *symmetry-breaking* paradigm that we mentioned initially: If all the processes operated in lockstep, repeatedly trying to access the database at the same time, there'd be no progress; but by randomizing, they "smooth out" the contention.

Analyzing the Algorithm

As with many applications of randomization, the algorithm in this case is extremely simple to state; the interesting issue is to analyze its performance.

Defining Some Basic Events When confronted with a probabilistic system like this, a good first step is to write down some basic events and think about their probabilities. Here's a first event to consider. For a given process P_i and a given round t, let $\mathcal{A}[i,t]$ denote the event that P_i attempts to access the database in round t. We know that each process attempts an access in each round with probability p, so the probability of this event, for any i and t, is $\Pr\left[\mathcal{A}[i,t]\right] = p$. For every event, there is also a *complementary event*, indicating that the event did not occur; here we have the complementary event $\overline{\mathcal{A}[i,t]}$ that P_i does not attempt to access the database in round t, with probability

$$\Pr\left[\overline{\mathcal{A}[i,t]}\right] = 1 - \Pr\left[\mathcal{A}[i,t]\right] = 1 - p.$$

Our real concern is whether a process *succeeds* in accessing the database in a given round. Let $\mathcal{S}[i,t]$ denote this event. Clearly, the process P_i must attempt an access in round t in order to succeed. Indeed, succeeding is equivalent to the following: Process P_i attempts to access the database in round t, and each other process *does not* attempt to access the database in round t. Thus $\mathcal{S}[i,t]$ is equal to the intersection of the event $\mathcal{A}[i,t]$ with all the complementary events $\overline{\mathcal{A}[j,t]}$, for $j \neq i$:

$$\mathcal{S}[i,t] = \mathcal{A}[i,t] \cap \left(\bigcap_{j \neq i} \overline{\mathcal{A}[j,t]} \right).$$

All the events in this intersection are independent, by the definition of the contention-resolution protocol. Thus, to get the probability of $\mathcal{S}[i,t]$, we can multiply the probabilities of all the events in the intersection:

$$\Pr\left[\mathcal{S}[i,t]\right] = \Pr\left[\mathcal{A}[i,t]\right] \cdot \prod_{j \neq i} \Pr\left[\overline{\mathcal{A}[j,t]}\right] = p(1-p)^{n-1}.$$

We now have a nice, closed-form expression for the probability that P_i succeeds in accessing the database in round t; we can now ask how to set p so that this success probability is maximized. Observe first that the success probability is 0 for the extreme cases $p = 0$ and $p = 1$ (these correspond to the extreme case in which processes never bother attempting, and the opposite extreme case in which every process tries accessing the database in every round, so that everyone is locked out). The function $f(p) = p(1-p)^{n-1}$ is positive for values of p strictly between 0 and 1, and its derivative $f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$ has a single zero at the value $p = 1/n$, where the maximum is achieved. Thus we can maximize the success probability by setting $p = 1/n$. (Notice that $p = 1/n$ is a natural intuitive choice as well, if one wants exactly one process to attempt an access in any round.)

When we set $p = 1/n$, we get $\Pr\left[\mathcal{S}[i,t]\right] = \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$. It's worth getting a sense for the asymptotic value of this expression, with the help of the following extremely useful fact from basic calculus.

(13.1)

(a) The function $\left(1-\frac{1}{n}\right)^n$ converges monotonically from $\frac{1}{4}$ up to $\frac{1}{e}$ as n increases from 2.

(b) The function $\left(1-\frac{1}{n}\right)^{n-1}$ converges monotonically from $\frac{1}{2}$ down to $\frac{1}{e}$ as n increases from 2.

Using (13.1), we see that $1/(en) \leq \Pr\left[\mathcal{S}[i,t]\right] \leq 1/(2n)$, and hence $\Pr\left[\mathcal{S}[i,t]\right]$ is asymptotically equal to $\Theta(1/n)$.

Waiting for a Particular Process to Succeed Let's consider this protocol with the optimal value $p = 1/n$ for the access probability. Suppose we are interested in how long it will take process P_i to succeed in accessing the database at least once. We see from the earlier calculation that the probability of its succeeding in any one round is not very good, if n is reasonably large. How about if we consider multiple rounds?

Let $\mathcal{F}[i,t]$ denote the "failure event" that process P_i does not succeed in *any* of the rounds 1 through t. This is clearly just the intersection of the complementary events $\overline{\mathcal{S}[i,r]}$ for $r = 1, 2, \ldots, t$. Moreover, since each of these events is independent, we can compute the probability of $\mathcal{F}[i,t]$ by multiplication:

$$\Pr\left[\mathcal{F}[i,t]\right] = \Pr\left[\bigcap_{r=1}^{t} \overline{\mathcal{S}[i,r]}\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] = \left[1-\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}\right]^t.$$

This calculation does give us the value of the probability; but at this point, we're in danger of ending up with some extremely complicated-looking expressions, and so it's important to start thinking asymptotically. Recall that the probability of success was $\Theta(1/n)$ after one round; specifically, it was bounded between $1/(en)$ and $1/(2n)$. Using the expression above, we have

$$\Pr\left[\mathcal{F}[i,t]\right] = \prod_{r=1}^{t} \Pr\left[\overline{\mathcal{S}[i,r]}\right] \leq \left(1-\frac{1}{en}\right)^t.$$

Now we notice that if we set $t = en$, then we have an expression that can be plugged directly into (13.1). Of course en will not be an integer; so we can take $t = \lceil en \rceil$ and write

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1-\frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1-\frac{1}{en}\right)^{en} \leq \frac{1}{e}.$$

This is a very compact and useful asymptotic statement: The probability that process P_i does not succeed in any of rounds 1 through $\lceil en \rceil$ is upper-bounded by the constant e^{-1}, independent of n. Now, if we increase t by some fairly small factors, the probability that P_i does not succeed in any of rounds 1 through t drops precipitously: If we set $t = \lceil en \rceil \cdot (c \ln n)$, then we have

$$\Pr\left[\mathcal{F}[i,t]\right] \leq \left(1 - \frac{1}{en}\right)^t = \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}.$$

So, asymptotically, we can view things as follows. After $\Theta(n)$ rounds, the probability that P_i has not yet succeeded is bounded by a constant; and between then and $\Theta(n \ln n)$, this probability drops to a quantity that is quite small, bounded by an inverse polynomial in n.

Waiting for All Processes to Get Through Finally, we're in a position to ask the question that was implicit in the overall setup: How many rounds must elapse before there's a high probability that all processes will have succeeded in accessing the database at least once?

To address this, we say that the protocol *fails* after t rounds if some process has not yet succeeded in accessing the database. Let $\mathcal{F}_t$ denote the event that the protocol fails after t rounds; the goal is to find a reasonably small value of t for which $\Pr\left[\mathcal{F}_t\right]$ is small.

The event $\mathcal{F}_t$ occurs if and only if one of the events $\mathcal{F}[i,t]$ occurs; so we can write

$$\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t].$$

Previously, we considered intersections of independent events, which were very simple to work with; here, by contrast, we have a union of events that are not independent. Probabilities of unions like this can be very hard to compute exactly, and in many settings it is enough to analyze them using a simple *Union Bound*, which says that the probability of a union of events is upper-bounded by the sum of their individual probabilities:

(13.2) (The Union Bound) *Given events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, we have*

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right].$$

Note that this is not an equality; but the upper bound is good enough when, as here, the union on the left-hand side represents a "bad event" that

we're trying to avoid, and we want a bound on its probability in terms of constituent "bad events" on the right-hand side.

For the case at hand, recall that $\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$, and so

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right].$$

The expression on the right-hand side is a sum of n terms, each with the same value; so to make the probability of $\mathcal{F}_t$ small, we need to make each of the terms on the right significantly smaller than $1/n$. From our earlier discussion, we see that choosing $t = \Theta(n)$ will not be good enough, since then each term on the right is only bounded by a constant. If we choose $t = \lceil en \rceil \cdot (c \ln n)$, then we have $\Pr\left[\mathcal{F}[i,t]\right] \leq n^{-c}$ for each i, which is what we want. Thus, in particular, taking $t = 2\lceil en \rceil \ln n$ gives us

$$\Pr\left[\mathcal{F}_t\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{F}[i,t]\right] \leq n \cdot n^{-2} = n^{-1},$$

and so we have shown the following.

(13.3) *With probability at least $1 - n^{-1}$, all processes succeed in accessing the database at least once within $t = 2\lceil en \rceil \ln n$ rounds.*

An interesting observation here is that if we had chosen a value of t equal to $qn \ln n$ for a very small value of q (rather than the coefficient $2e$ that we actually used), then we would have gotten an upper bound for $\Pr\left[\mathcal{F}[i,t]\right]$ that was larger than n^{-1}, and hence a corresponding upper bound for the overall failure probability $\Pr\left[\mathcal{F}_t\right]$ that was larger than 1—in other words, a completely worthless bound. Yet, as we saw, by choosing larger and larger values for the coefficient q, we can drive the upper bound on $\Pr\left[\mathcal{F}_t\right]$ down to n^{-c} for any constant c we want; and this is really a very tiny upper bound. So, in a sense, all the "action" in the Union Bound takes place rapidly in the period when $t = \Theta(n \ln n)$; as we vary the hidden constant inside the $\Theta(\cdot)$, the Union Bound goes from providing no information to giving an extremely strong upper bound on the probability.

We can ask whether this is simply an artifact of using the Union Bound for our upper bound, or whether it's intrinsic to the process we're observing. Although we won't do the (somewhat messy) calculations here, one can show that when t is a small constant times $n \ln n$, there really is a sizable probability that some process has not yet succeeded in accessing the database. So a rapid falling-off in the value of $\Pr\left[\mathcal{F}_t\right]$ genuinely does happen over the range $t = \Theta(n \ln n)$. For this problem, as in many problems of this flavor, we're

really identifying the asymptotically "correct" value of t despite our use of the seemingly weak Union Bound.

13.2 Finding the Global Minimum Cut

Randomization naturally suggested itself in the previous example, since we were assuming a model with many processes that could not directly communicate. We now look at a problem on graphs for which a randomized approach comes as somewhat more of a surprise, since it is a problem for which perfectly reasonable deterministic algorithms exist as well.

The Problem

Given an undirected graph $G = (V, E)$, we define a *cut* of G to be a partition of V into two non-empty sets A and B. Earlier, when we looked at network flows, we worked with the closely related definition of an *s-t cut*: there, given a directed graph $G = (V, E)$ with distinguished source and sink nodes s and t, an *s-t cut* was defined to be a partition of V into sets A and B such that $s \in A$ and $t \in B$. Our definition now is slightly different, since the underlying graph is now undirected and there is no source or sink.

For a cut (A, B) in an undirected graph G, the *size* of (A, B) is the number of edges with one end in A and the other in B. A *global minimum cut* (or "global min-cut" for short) is a cut of minimum size. The term *global* here is meant to connote that any cut of the graph is allowed; there is no source or sink. Thus the global min-cut is a natural "robustness" parameter; it is the smallest number of edges whose deletion disconnects the graph. We first check that network flow techniques are indeed sufficient to find a global min-cut.

(13.4) *There is a polynomial-time algorithm to find a global min-cut in an undirected graph G.*

Proof. We start from the similarity between cuts in undirected graphs and *s-t* cuts in directed graphs, and with the fact that we know how to find the latter optimally.

So given an undirected graph $G = (V, E)$, we need to transform it so that there are directed edges and there is a source and sink. We first replace every undirected edge $e = (u, v) \in E$ with two oppositely oriented directed edges, $e' = (u, v)$ and $e'' = (v, u)$, each of capacity 1. Let G' denote the resulting directed graph.

Now suppose we pick two arbitrary nodes $s, t \in V$, and find the minimum *s-t* cut in G'. It is easy to check that if (A, B) is this minimum cut in G', then (A, B) is also a cut of minimum size in G *among all those that separate s from t*. But we know that the global min-cut in G must separate s from *something*,