performance. The randomized sorting algorithm described above is an example. This book presents many other randomized algorithms that enjoy these advantages.

In the next few sections, we will illustrate some basic ideas from probability theory using simple applications to randomized algorithms. The reader wishing to review some of the background material on the analysis of algorithms or on elementary probability theory is referred to the Appendices.

1.1. A Min-Cut Algorithm

Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are said to be *independent* if the probability that they both occur is given by

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2] \tag{1.4}$$

(see Appendix C). In the more general case where $\mathcal{E}_1$ and $\mathcal{E}_2$ are not necessarily independent,

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2] \times \mathbf{Pr}[\mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_1], \tag{1.5}$$

where $\mathbf{Pr}[\mathcal{E}_1 \mid \mathcal{E}_2]$ denotes the *conditional probability* of $\mathcal{E}_1$ given $\mathcal{E}_2$. Sometimes, when a collection of events is not independent, a convenient method for computing the probability of their intersection is to use the following generalization of (1.5).

$$\mathbf{Pr}[\cap_{i=1}^{k} \mathcal{E}_i] = \mathbf{Pr}[\mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \times \mathbf{Pr}[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] \cdots \mathbf{Pr}[\mathcal{E}_k \mid \cap_{i=1}^{k-1} \mathcal{E}_i]. \tag{1.6}$$

Consider a graph-theoretic example. Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality. We now study a simple algorithm for finding a min-cut of a graph.

We repeat the following step: pick an edge uniformly at random and merge the two vertices at its end-points (Figure 1.1). If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. With each contraction, the number of vertices of G decreases by one. The crucial observation is that an edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph. The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut.

Does this algorithm always find a min-cut? Let us analyze its behavior after first reviewing some elementary definitions from graph theory.

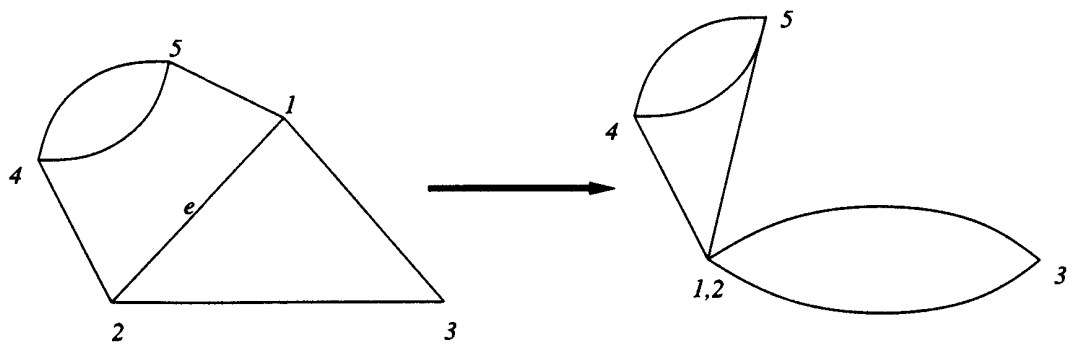

Figure 1.1: A step in the min-cut algorithm; the effect of contracting edge $e = (1,2)$ is shown.

► **Definition 1.1:** For any vertex v in a multigraph G, the *neighborhood* of G, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Note that $d(v)$ is the same as the cardinality of $\Gamma(v)$ when there are no self-loops or multiple edges between v and any of its neighbors.

Let k be the min-cut size. We fix our attention on a particular min-cut C with k edges. Clearly G has at least $kn/2$ edges; otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k. We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

Let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step, for $1 \leq i \leq n-2$. The probability that the edge randomly chosen in the first step is in C is at most $k/(nk/2) = 2/n$, so that $\mathbf{Pr}[\mathcal{E}_1] \geq 1-2/n$. Assuming that $\mathcal{E}_1$ occurs, during the second step there are at least $k(n-1)/2$ edges, so the probability of picking an edge in C is at most $2/(n-1)$, so that $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1-2/(n-1)$. At the ith step, the number of remaining vertices is $n-i+1$. The size of the min-cut is still at least k, so the graph has at least $k(n-i+1)/2$ edges remaining at this step. Thus, $\mathbf{Pr}[\mathcal{E}_i \mid \cap_{j=1}^{i-1}\mathcal{E}_j] \geq 1-2/(n-i+1)$. What is the probability that no edge of C is ever picked in the process? We invoke (1.6) to obtain

$$\mathbf{Pr}[\cap_{i=1}^{n-2}\mathcal{E}_i] \geq \prod_{i=1}^{n-2}\left(1-\frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

The probability of discovering a particular min-cut (which may in fact be the unique min-cut in G) is larger than $2/n^2$. Thus our algorithm may err in declaring the cut it outputs to be a min-cut. Suppose we were to repeat the above algorithm $n^2/2$ times, making independent random choices each time. By (1.4), the probability that a min-cut is not found in any of the $n^2/2$

attempts is at most

$$\left(1-\frac{2}{n^2}\right)^{n^2/2} < 1/e.$$

By this process of repetition, we have managed to reduce the probability of failure from $1-2/n^2$ to a more respectable $1/e$. Further executions of the algorithm will make the failure probability arbitrarily small – the only consideration being that repetitions increase the running time.

Note the extreme simplicity of the randomized algorithm we have just studied. In contrast, most deterministic algorithms for this problem are based on network flows and are considerably more complicated. In Section 10.2 we will return to the min-cut problem and fill in some implementation details that have been glossed over in the above presentation; in fact, it will be shown that a variant of this algorithm has an expected running time that is significantly smaller than that of the best known algorithms based on network flow.

Exercise 1.2: Suppose that at each step of our min-cut algorithm, instead of choosing a random edge for contraction we choose two vertices at random and coalesce them into a single vertex. Show that there are inputs on which the probability that this modified algorithm finds a min-cut is exponentially small.

1.2. Las Vegas and Monte Carlo

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm *always* gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. We call such an algorithm a *Las Vegas algorithm.*

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we are able to bound the probability of such an incorrect solution. We call such an algorithm a *Monte Carlo algorithm.* In Section 1.1 we observed a useful property of a Monte Carlo algorithm: if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. Later, we will see examples of algorithms in which both the running time and the quality of the solution are random variables; sometimes these are also referred to as Monte Carlo algorithms. For decision problems (problems for which the answer to an instance is YES or NO), there are two kinds of Monte Carlo algorithms: those with *one-sided error*, and those with *two-sided error.* A Monte Carlo algorithm is said to have two-sided error if there is a non-zero probability that it errs when it outputs either YES or NO. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs (YES/NO) that it produces.

We will see examples of all three types of algorithms – Las Vegas, Monte Carlo with one-sided error, and Monte Carlo with two-sided error – in this book.

Which is better, Monte Carlo or Las Vegas? The answer depends on the application – in some applications an incorrect solution may be catastrophic. A Las Vegas algorithm is by definition a Monte Carlo algorithm with error probability 0. The following exercise gives us a way of deriving a Las Vegas algorithm from a Monte Carlo algorithm. Note that the efficiency of the derivation procedure depends on the time taken to verify the correctness of a solution to the problem.

Exercise 1.3: Consider a Monte Carlo algorithm A for a problem Π whose expected running time is at most $T(n)$ on any instance of size n and that produces a correct solution with probability $\gamma(n)$. Suppose further that given a solution to Π, we can verify its correctness in time $t(n)$. Show how to obtain a Las Vegas algorithm that always gives a correct answer to Π and runs in expected time at most $(T(n) + t(n))/\gamma(n)$.

In attempting Exercise 1.3 the reader will have to use a simple property of the *geometric random variable* (Appendix C). Consider a biased coin that, on a toss, has probability p of coming up HEADS and $1 - p$ of coming up TAILS. What is the expected number of (independent) tosses up to and including the first head? The number of such tosses is a random variable that is said to be *geometrically distributed.* The expectation of this random variable is $1/p$. This fact will prove useful in numerous applications.

Exercise 1.4: Let $0 < \epsilon_2 < \epsilon_1 < 1$. Consider a Monte Carlo algorithm that gives the correct solution to a problem with probability at least $1 - \epsilon_1$, regardless of the input. How many independent executions of this algorithm suffice to raise the probability of obtaining a correct solution to at least $1 - \epsilon_2$, regardless of the input?

We say that a Las Vegas algorithm is an *efficient Las Vegas* algorithm if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say that a Monte Carlo algorithm is an *efficient Monte Carlo* algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size.

1.3. Binary Planar Partitions

We now illustrate another very useful and basic tool from probability theory: *linearity of expectation.* For random variables $X_1, X_2, \ldots$,

$$\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i]. \tag{1.7}$$