22 Monte Carlo Randomized Algorithms

In the last chapter we studied randomized algorithms of the Las Vegas variety. This chapter is devoted to randomized algorithms of the Monte Carlo variety.

A **Monte Carlo** algorithm typically runs in a fixed amount of time, where the runtime is typically independent of the random choices made. However, it only produces the correct answer some fraction of the time (say half the time). The error probability depends on the particular random bits. Hence, runs are independent of each other, and one can improve the correctness by running the algorithm multiple times (with different sequences of random bits).

This definition is very abstract, so let's turn to some common examples.

22.1 Randomized Matrix-Multiplication Checking

One of the most common uses of randomized algorithms is to verify the correctness of a program, a.k.a. **program checking**. The typical scenario is that one has a program that one doesn't totally trust. One would like to check the correctness of the output very quickly – in way less time than it would take to run the computation from scratch.

As an example, consider the problem of multiplying two matrices.

Question: Using standard matrix multiplication, how many multiplications are needed to multiply two $n \times n$ matrices, $\mathbf{A}$ and $\mathbf{B}$?

Answer: $\Theta(n^3)$.

Question: How many multiplications are needed using the currently fastest method for multiplying matrices?

Answer: You've probably heard of Strassen's algorithm, which uses $O(n^{\log_2 7 + o(1)}) \approx O(n^{2.8})$ multiplications. Until recently, the fastest algorithm was due to Coppersmith and Winograd [15], using $O(n^{2.376})$ multiplications.

Recently, a CMU PhD student, Virginia Vassilevska, improved this to $O(n^{2.373})$ multiplications [79].

The Coppersmith and Winograd (C-W) algorithm is complex to implement, and the Vassilevska algorithm is even more so. Suppose that someone has an implementation of the Vassilevska algorithm. You might not trust their code.

You would ideally like to be able to check each output that the Vassilevska implementation gives you. That is, every time that you input two $n \times n$ matrices, $\mathbf{A}$ and $\mathbf{B}$, into the Vassilevska implementation, and it outputs $\mathbf{C}$, you'd like to check if in fact $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$.

Of course we could check that $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ using standard matrix multiplication. But this would take $\Theta(n^3)$ time, which would defeat the whole point of using the $\Theta(n^{2.37})$ implementation.

We now illustrate a randomized algorithm due to Freivalds [29] that allows us to check whether $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ using only $\Theta(n^2)$ multiplications. Hence, we can afford to run our checker every time that we run the Vassilevska implementation.

Algorithm 22.1 (Freivalds' matrix multiplication checking algorithm)
Inputs: $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ *all of dimension* $n \times n$, *with elements in* $\mathbb{R}$.

1. *Choose a random vector* $\vec{r} = (r_1, r_2, r_3, \ldots, r_n)$, *where* $r_i \in \{0, 1\}$.
2. *Compute* $\mathbf{B}\vec{r}$.
3. *Compute* $\mathbf{A}(\mathbf{B}\vec{r})$.
4. *Compute* $\mathbf{C}\vec{r}$.
5. *If* $\mathbf{A}(\mathbf{B}\vec{r}) \neq \mathbf{C}\vec{r}$, *then return:* $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$. *Otherwise, return* $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$.

Question: How many multiplications are needed by Freivalds' algorithm?

Answer: Each of steps 2, 3, and 4 only involve multiplication of a matrix by a vector, and hence use only $\Theta(n^2)$ multiplications.

Question: If Freivalds' algorithm returns $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, is it correct?

Answer: Yes. Suppose, by contradiction, that $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. Then, $\forall \vec{r}$, it must be the case that $\mathbf{A} \cdot \mathbf{B} \cdot \vec{r} = \mathbf{C} \cdot \vec{r}$.

Definition 22.2 *When* $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, *and* $\vec{r}$ *is a vector such that*

$$\mathbf{A} \cdot \mathbf{B} \cdot \vec{r} \neq \mathbf{C} \cdot \vec{r},$$

then we say that $\vec{r}$ *is a* **witness** *to the fact that* $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, *because it provides us with a proof that* $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$.

Thus the only time that Freivalds' algorithm might be wrong is if it returns $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, even though $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$. This is referred to as **one-sided error**. Theorem 22.3 shows that the probability of this type of mistake is $\leq \frac{1}{2}$, given that $\vec{r}$ is chosen at random.

Theorem 22.3 (Freivalds error) *Let* $\mathbf{A}$, $\mathbf{B}$, *and* $\mathbf{C}$ *denote* $n \times n$ *matrices, where* $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, *and let* $\vec{r}$ *be a vector chosen uniformly at random from* $\{0,1\}^n$. *Then,*

$$\mathbf{P}\{\mathbf{A} \cdot \mathbf{B} \cdot \vec{r} = \mathbf{C} \cdot \vec{r}\} \leq \frac{1}{2}.$$

Proof: [Theorem 22.3] Since $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, we know that

$$\mathbf{D} \equiv \mathbf{AB} - \mathbf{C} \neq \mathbf{O},$$

where $\mathbf{O}$ is an $n \times n$ matrix of all zeros.

Now suppose that $\vec{r}$ is a vector such that

$$\mathbf{D}\vec{r} = \mathbf{AB}\vec{r} - \mathbf{C}\vec{r} = \vec{0},$$

as shown here:

$$\begin{bmatrix} d_{11} & d_{12} & d_{13} & \dots & d_{1n} \\ d_{21} & d_{22} & d_{23} & \dots & d_{2n} \\ d_{31} & d_{32} & d_{33} & \dots & d_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{n1} & d_{n2} & d_{n3} & \dots & d_{nn} \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Since $\mathbf{D} \neq \mathbf{O}$, we know that $\mathbf{D}$ must have at least one non-zero entry. For notational convenience, we will assume that this non-zero entry is d_{11} (you will see that this assumption is made without loss of generality).

Since $\mathbf{D}\vec{r} = \vec{0}$, we know that the product of the first row of $\mathbf{D}$ and $\vec{r}$ yields 0, that is,

$$\sum_{j=1}^{n} d_{1j} \cdot r_j = 0.$$

But this implies that

$$d_{11}r_1 + \sum_{j=2}^{n} d_{1j}r_j = 0$$

$$\Rightarrow r_1 = -\frac{\sum_{j=2}^{n} d_{1j}r_j}{d_{11}}. \tag{22.1}$$

Recall that $d_{11} \neq 0$ so the denominator of (22.1) is non-zero.

Question: Does (22.1) imply that r_1 is negative?

Answer: No, d_{1j} may be negative.

Now suppose that when choosing the random vector $\vec{r}$, we choose $r_2, \ldots, r_n$ *before* choosing r_1. Consider the moment just after we have chosen $r_2, \ldots, r_n$. At this moment the right-hand side of (22.1) is determined. Thus there is *exactly one choice* for r_1 (call this r_1^*) that will make (22.1) true.

Question: We now flip a 0/1 coin to determine r_1. What is $\mathbf{P}\left\{r_1 = r_1^*\right\}$?

Answer: There are two possible values for r_1 (namely 0 or 1). They can't both be equal to r_1^*. Thus, $\mathbf{P}\left\{r_1 = r_1^*\right\} \leq \frac{1}{2}$. Note this is not an equality because r_1^* can be any element of the Reals and thus is not necessarily $\in \{0, 1\}$. ■

At this point we have proven that Freivalds' algorithm can check matrix multiplication in $\Theta(n^2)$ time with accuracy of at least $\frac{1}{2}$.

Question: How can we improve the accuracy?

Hint: Repeat Freivalds' algorithm with additional random vectors, $\vec{r} \in \{0, 1\}^n$.

Answer: Suppose we run Freivalds' algorithm using k randomly chosen $\vec{r}$ vectors. If for any vector $\vec{r}$ we see that $\mathbf{A}(\mathbf{B}\vec{r}) \neq \mathbf{C}\vec{r}$, then we output $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$; otherwise we output $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. If in fact $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, then all k runs will output "equal," and we will have the correct answer at the end. If $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$, then each run has independent probability $\geq \frac{1}{2}$ of discovering ("witnessing") this fact. Thus the probability that it is not discovered that $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{C}$ is $\leq \frac{1}{2^k}$. The k runs require a total runtime of $\Theta(kn^2)$. If we make $k = 100$, yielding an extremely low probability of error, our overall runtime for using the Vassilevska implementation is still $\Theta(n^{2.37} + 100n^2) = \Theta(n^{2.37})$.

Question: Is it a problem if some of the random vectors $\vec{r}$ repeat? Does this change our confidence in the final answer?

Answer: This is not a problem. All that's needed is that the vectors are picked independently. Each time we pick a random $\vec{r}$, that choice will independently have probability $\geq \frac{1}{2}$ of being a witness.

Question: Suppose that the error in Freivalds' algorithm was not one-sided, but rather two-sided? Would we still be able to use this scheme?

Answer: The exact scheme we're using assumes one-sided error. However, we could use a related scheme in the case of two-sided error, where we take the "majority" output of several runs.

Question: Can we improve our confidence by choosing $\vec{r} \in \{0,1,2\}^n$?

Answer: Yes, this drops the probability of error to $\frac{1}{3}$ for each Freivalds check because, again, there is exactly one value of r_1^* that allows us to mess up, and our chance of now hitting that value is $\leq \frac{1}{3}$.

22.2 Randomized Polynomial Checking

We now apply a very similar idea to the question of multiplication of monomials over some real-valued variable x. Suppose we have a program that purports to multiply together monomials. For example, our program might take as input a string of three monomials:

$$(x-3)\ ,\ (x-5)\ ,\ (x+7)$$

and output the third degree polynomial:

$$G(x) = x^3 - x^2 - 41x + 105.$$

Again, we'd like a way of *checking* the output of this multiplication program very quickly, in much less time than the runtime of the program. Here "runtime" refers to the number of multiplications.

Throughout, let's define $F(x)$ to be the (true) product of the monomials, that is,

$$F(x) \equiv (x-3)(x-5)(x+7),$$

whereas $G(x)$ represents the output of our untrusted program. The goal of the checker is to determine whether $G(x)$ is equal to the product $(x-3)(x-5)(x+7)$ without computing $F(x)$.

Question: How many multiplications are needed to multiply d monomials?

Answer: $O(d^2)$. See Exercise 22.3.

We will now present a randomized checker that determines with high certainty whether our untrusted program is correct using only $\Theta(d)$ multiplications. As before, we start with a simple checker that makes mistakes, and then we improve the checker to lower its probability of mistakes.

Algorithm 22.4 (Simple Checker for multiplication of d monomials)

1. *Pick an integer, r, uniformly at random between* 1 *and* $100d$.
2. *Evaluate $F(r)$. Evaluate $G(r)$.*
3. *If $F(r) = G(r)$, then output that the program is correct. Otherwise, output that it is incorrect.*

Question: What is the runtime of our Simple Checker?

Answer: $\Theta(d)$. Once you replace x by r, there are only d multiplications needed.

Question: Under what condition is the Simple Checker wrong?

Answer: If $F(x) = G(x)$, then the Simple Checker will always be correct. If $F(x) \neq G(x)$, then the Simple Checker might make an error if it only picks r values for which $F(r) = G(r)$. Again, this is a situation of **one-sided error**.

Question: So what is the probability that the Simple Checker is wrong?

Hint: If $F(x) \neq G(x)$, and each are polynomials of degree d, what is the maximum number of values of x on which they can nonetheless agree?

Answer: Assume that $F(x) \neq G(x)$. Let $H(x) = F(x) - G(x)$. $H(x)$ is at most a d-degree polynomial. As such, it has at most d roots (assuming $H(x) \neq 0$). Hence, there are at most d possible values of r, s.t. $H(r) = 0$. Equivalently, there are at most d possible values of r on which $F(x)$ and $G(x)$ agree. If we now limit the range of r from 1 to $100d$, then there are still at most d values of r for which $F(r) = G(r)$. Hence, the probability that we have found such an r is at most $\frac{d}{100d} = \frac{1}{100}$.

So our Simple Checker accurately tells us whether our program is correct with probability $\frac{99}{100}$ in only $\Theta(d)$ time.

Question: Suppose we'd like to know that our program is correct with probability $\frac{999{,}999}{1{,}000{,}000}$? How can we modify our Simple Checker to get this higher guarantee?

Answer: One idea is to pick our random r from a bigger range. For example, we can use a range from 1 to $10^6 d$. However, this may not be feasible for large d.

Answer: A better idea is to repeat our Simple Checker with different values of r. This will give higher accuracy, but require more time.

Algorithm 22.5 (Superior Checker)

1. *Repeat the Simple Checker k times, each time with some r drawn uniformly at random from the set of integers in* $[1, 100d]$.
2. *If $F(r) = G(r)$ for* all *k values of r, then output that the program is correct. Otherwise, output that the program is wrong.*

Question: What is the probability that the Superior Checker is wrong?

Answer: The Superior Checker only fails if $F(x) \neq G(x)$ and yet $F(r) = G(r)$ for all k values of r. But each time we draw a random r, that r has probability

$\geq \frac{99}{100}$ of resulting in inequality. So the probability that all k values of r result in equality is $\leq \frac{1}{100^k}$. With just $k = 3$, we already have our $\frac{999{,}999}{1{,}000{,}000}$ confidence.

Observe that the runtime of the Superior Checker is only $\Theta(kd)$.

22.3 Randomized Min-Cut

Throughout this section, we assume that we have an undirected graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

Definition 22.6 *A* **cut-set** *of a graph $G = (V, E)$ is a set of edges whose removal breaks the graph into two or more connected components.*

Definition 22.7 *A* **min-cut** *is a minimum cardinality cut set.*

The **Min-cut Problem** is the problem of finding a **min-cut**. Observe that there may be several minimum cardinality cut sets possible – we just want one of them. The Min-Cut problem has many applications, mostly dealing with reliability. For example, what is the minimum number of links that can fail before the network becomes disconnected?

We will now present a randomized algorithm for finding a min-cut that is both faster and simpler than any deterministic algorithm. Our algorithm is based on the idea of "contracting edges" until a cut-set results.

Definition 22.8 *The* **contraction of an edge** (v_1, v_2) *involves merging vertices v_1 and v_2 into a single vertex, v. Any edge (v_1, v_2) is removed. All edges that had an endpoint at either v_1 or v_2 (but not both) will now have an endpoint at the contracted vertex, v. Observe that the new graph may have parallel edges, but no self-loops.*

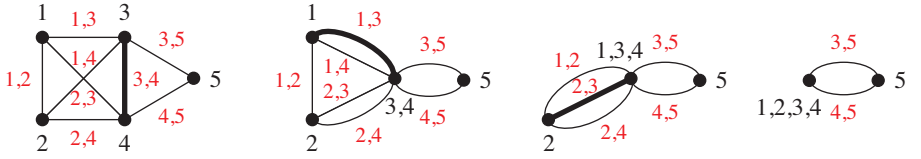
Algorithm 22.9 (Randomized Min-Cut algorithm)

1. *Given a graph, G, repeat until only two vertices, u and v, remain:*
   i. *Pick a random edge from all existing edges in the graph.*
   ii. *Contract that edge.*
2. *Output the set of edges connecting u and v.*

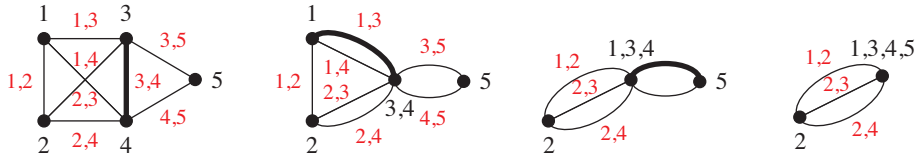
Question: How many iterations are needed by the Randomized Min-Cut algorithm?

Answer: $n - 2$.

Figure 22.1 shows one example of Randomized Min-Cut that results in a min-cut and another that doesn't. In the example that works, the cut that is output should be interpreted as the two edges between vertex 5 and other vertices in the graph (looking at the original graph, we see that vertex 5 is only connected to vertices 3 and 4). In the example that doesn't work, the cut that is output should be interpreted as the three edges between vertex 2 and the other vertices in the graph (note that vertex 2 is only connected to vertices 1, 3, and 4 in the original graph).



(a) Successful run of Randomized Min-Cut



(b) Unsuccessful run of Randomized Min-Cut

Figure 22.1 *Example of two runs of Randomized Min-Cut. The bold edge is the one about to be contracted.*

Question: Let $G = (V, E)$ refer to the original graph. Which, if any, of the following statements is true?

(a) Any cut-set of an intermediate graph is also a cut-set of G.
(b) Any cut-set of G is also a cut-set of every intermediate graph.

Answer: The first statement is true. Let C be a cut-set in an intermediate graph that separates vertices in S from those in $V - S$. All edges in C are edges from the original graph, $G = (V, E)$. Now suppose that there were additional edges between S and $V - S$ in the original graph, $G = (V, E)$. This couldn't happen because as soon as one of those was contracted away, then S and $V - S$ couldn't be separated. So C still forms a cut-set in G. The second statement is clearly false, as seen in Figure 22.1. The issue is that some edges of the cut-set of the original graph may have been contracted away.

So the output of Randomized Min-Cut is always a true cut-set of the original graph, but not necessarily a minimally sized cut-set.

We now state one more property of Randomized Min-Cut that will be useful in its analysis.

Lemma 22.10 *Let C be a cut-set of graph $G = (V, E)$. Let k be the cardinality of C. Suppose we run Randomized Min-Cut for the full $n - 2$ iterations. Then,*

$$C \text{ is output by Min-Cut} \iff \text{None of the } k \text{ edges of } C \text{ are contracted.}$$

Proof:
($\Longrightarrow$) This direction is obvious in that C cannot be output if any of its edges are contracted.

($\Longleftarrow$) This direction is much less clear. For one thing, could it be that the cut that is output includes C *plus* some additional edge? For another thing, could it be that the cut that is output is *missing* some edge, $\vec{e}$, of C, because $\vec{e}$ got removed when a *parallel edge* to $\vec{e}$ got contracted?

We now address both issues. Suppose that C splits the graph into two components, called A and $\overline{A}$, as shown in Figure 22.2. Let E denote all edges in G. Let E_A, and $E_{\overline{A}}$ denote the set of edges in A, and $\overline{A}$, respectively. Hence $E = C \cup E_A \cup E_{\overline{A}}$.

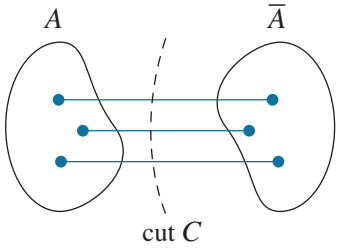


Figure 22.2 *By definition of C being a cut, we must have $E = E_C \cup E_A \cup E_{\overline{A}}$.*

Since cut C already splits graph G into A and $\overline{A}$, by definition there cannot be any "additional edges" beyond C that have one endpoint in A and one in $\overline{A}$. Thus it can't be the case that the final cut that is output includes C plus some additional edges. Likewise, since all edges outside of C must be in either A or $\overline{A}$, then contracting an edge outside of set C cannot result in some edge of C getting removed. ■

Theorem 22.11 *The Randomized Min-Cut algorithm produces a min-cut with probability $\geq \frac{2}{n(n-1)}$.*

Proof: Let C be one of the min-cuts in the original graph, $G = (V, E)$. Let the cardinality of C be k. We will show that with probability $\geq \frac{2}{n(n-1)}$, C will be output by our algorithm.

By Lemma 22.10, the probability that C is output at the end of our algorithm is the probability that none of the k edges of C is contracted during the $n-2$ iterations of the Randomized Min-Cut algorithm. To figure out this probability, it helps to first derive the number of edges in G.

Question: What is the minimum degree of vertices in G?

Answer: The minimum degree is k, because if some vertex had degree $< k$ then those edges would form a smaller cut-set.

Question: What is a lower bound on the number of edges in G?

Answer: G has at least $\frac{nk}{2}$ edges, from which edges are selected uniformly at random for contraction.

$$\mathbf{P}\{\text{no edge of } C \text{ is selected in the 1st round}\} \geq \frac{\frac{nk}{2} - k}{\frac{nk}{2}} = \frac{nk - 2k}{nk} = \frac{n-2}{n}.$$

Question: Why did we need a $\geq$ sign above?

Answer: Recall that $\frac{nk}{2}$ is a *lower bound* on the number of edges.

Suppose that after the first round, we did not eliminate an edge of C. We are left with a graph on $n-1$ vertices. The graph still has a min-cut of C.

Question: Why does the graph still have a min-cut of C?

Answer: Any cut-set of the contracted graph is also a cut-set of the original graph. So if the graph has a min-cut smaller than $|C|$, then the original graph must have a cut-set smaller than $|C|$, which is a contradiction.

Since the contracted graph still has a cut-set of C, the graph must have $\geq \frac{k(n-1)}{2}$ edges. Given this lower bound on the number of edges, we have:

$$\mathbf{P}\{\text{no edge of } C \text{ is selected in the 2nd round} \mid \text{no edge selected in 1st round}\}$$
$$\geq \frac{\frac{(n-1)k}{2} - k}{\frac{(n-1)k}{2}} = \frac{(n-1)k - 2k}{(n-1)k} = \frac{n-3}{n-1}.$$

Generalizing, let E_i be the event that no edge of C is selected in the ith round. We want the probability that all of the first $n-2$ events happen. By Theorem 2.10,

$$\mathbf{P}\{E_1 \cap E_2 \cap \cdots \cap E_{n-2}\} = \mathbf{P}\{E_1\} \cdot \mathbf{P}\{E_2 \mid E_1\} \cdots \mathbf{P}\left\{E_{n-2} \mid \cap_{i=1}^{n-1} E_i\right\}.$$

We have already shown that:

$$\mathbf{P}\{E_1\} \geq \frac{n-2}{n} \qquad \text{and} \qquad \mathbf{P}\{E_2 \mid E_1\} \geq \frac{n-3}{n-1}.$$

Via the same argument, we can see that:

$$\mathbf{P}\{E_3 \mid E_1 \cap E_2\} \geq \frac{n-4}{n-2},$$

and so on. Hence we have:

$$\begin{aligned}
&\mathbf{P}\{\text{No edge of } C \text{ is ever contracted}\} \\
&= \mathbf{P}\{E_1 \cap E_2 \cap \cdots \cap E_{n-2}\} \\
&= \mathbf{P}\{E_1\} \cdot \mathbf{P}\{E_2 \mid E_1\} \cdot \mathbf{P}\{E_3 \mid E_1 \cap E_2\} \cdots \mathbf{P}\left\{E_{n-2} \mid \cap_{i=1}^{n-3} E_i\right\} \\
&\geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{\cancel{n-2}}{n} \cdot \frac{\cancel{n-3}}{n-1} \cdot \frac{\cancel{n-4}}{\cancel{n-2}} \cdot \frac{\cancel{n-5}}{\cancel{n-3}} \cdots \frac{\cancel{3}}{\cancel{5}} \cdot \frac{2}{\cancel{4}} \cdot \frac{1}{\cancel{3}} \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

■

Observe that our algorithm has a high probability, $1 - \frac{2}{n(n-1)}$, of returning a cut-set that is *not* a min-cut.

Question: What can we do to reduce the probability that our algorithm is wrong?

Answer: We should run our algorithm many times and return the smallest cut-set produced by all those runs.

Claim 22.12 *If we run Randomized Min-Cut $\Theta(n^2 \ln n)$ times, and report the smallest cardinality cut-set returned, then with probability $> 1 - \frac{1}{n^2}$ our reported cut-set is a min-cut.*

Proof: The probability that our output is not a min-cut is upper-bounded by

$$\begin{aligned}
\left(1 - \frac{2}{n(n-1)}\right)^{n^2 \ln n} &\leq \left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\ln n} \\
&= \left[\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)}\right]^{\ln n} \\
&< \left[e^{-2}\right]^{\ln n} \\
&= \frac{1}{n^2},
\end{aligned}$$

where we've used the fact that $1 - x < e^{-x}$, for $0 < x < 1$, by (1.14). ■

Thus the total runtime of Randomized Min-Cut involves $\Theta(n^3 \ln n)$ contractions, which is still a very good runtime. Randomized Min-Cut also excels in its simplicity. There is of course some (tiny) possibility of error. However, here this error is not fatal in that the algorithm always gives a cut-set.

22.4 Related Readings

The Randomized Min-Cut algorithm is due to David Karger [42] and is famous for its simplicity. Since then many algorithms have been proposed with improved runtimes; see, for example, the Karger–Stein algorithm [43], which improves the runtime to $O(n^2(\log n)^3)$ time.

22.5 Exercises

22.1 **From Las Vegas to Monte Carlo and back**
In this problem, you will show how to take a Las Vegas randomized algorithm and make it into a Monte Carlo algorithm, and vice-versa.
(a) Given a Las Vegas algorithm, A_{LV}, that solves a problem P in *expected* time t, describe a Monte Carlo algorithm for problem P that runs in time *at most* $50t$ and outputs the correct answer with probability at least 98%.
(b) Can you do the reverse? Suppose you have a Monte Carlo algorithm, A_{MC}, that runs in time *at most* t_{MC} and gives the correct answer with probability p. Suppose that there's a verification routine, V, that tells us with perfect accuracy whether A_{MC} is correct in time t_V. Can you use this to design a Las Vegas algorithm, A_{LV}? If so, what is the expected runtime of A_{LV}?

22.2 **Freivalds error**
In the proof of Theorem 22.3, we defined a matrix $\mathbf{D} \neq \mathbf{O}$, and stated without loss of generality that $d_{11} \neq 0$. Suppose that instead $d_{22} \neq 0$. Rewrite the proof of Theorem 22.3 to show that it holds here as well.

22.3 **Multiplication of monomials**
Let a "monomial in x" be an expression of the form $(x-c)$, where c is some constant. Prove that the number of multiplications needed to multiply d monomials in x is $O(d^2)$. Note that the problem is not saying that d^2 multiplications suffice. You might need $10d^2$ multiplications. You'll need to figure out the appropriate multiplier for d^2. [Hint: Use induction.]

22.4 **Randomized polynomial checking with knowledge**
Let $F(x) \equiv (x - c_1)(x - c_2)(x - c_3) \cdots (x - c_d)$ be the true product of d monomials. Let $G(x)$ be a degree d polynomial which is claimed to equal $F(x)$. Recall the Simple Randomized Checker from Algorithm 22.4. Now suppose you know a priori that the coefficients in $G(x)$ for $x^d, x^{d-1}, x^{d-2}, \ldots, x^{d/2+1}$ are all correct (say, someone else has checked those for you). Assume that d is even. Provide an upper bound on the probability of error of the Simple Randomized Checker.

22.5 **Randomized Max-3SAT**
You have a bunch of variables: $W, X, Y, Z, \ldots$ Each variable is allowed to be either 0 or 1. The term $\overline{X}$ denotes the negation of X. A *clause* is the OR of three *distinct* variables. For example: (X **or** Y **or** $\overline{Z}$). A clause is *satisfied* if it evaluates to 1. For example, under the assignment: $X = 1$, $Y = 0$, $Z = 0$, (X **or** Y **or** $\overline{Z}$) evaluates to 1, and hence is satisfied.
A *3SAT expression* is the AND of a bunch of clauses: For example:

$$(X \textbf{ or } Y \textbf{ or } \overline{Z}) \textbf{ AND } (W \textbf{ or } \overline{X} \textbf{ or } \overline{Z}) \textbf{ AND } (Y \textbf{ or } \overline{X} \textbf{ or } Z).$$

The goal of Max-3SAT is to find an assignment of the variables that *maximizes* the number of satisfied clauses. For example, if we set $W = X = Y = Z = 1$ in the above 3SAT expression, then all three clauses will be satisfied. Max-3SAT is known to be NP-Hard.

(a) Propose a very simple Monte Carlo randomized algorithm for Max-3SAT. Let N denote the number of clauses satisfied by your algorithm. Show that $\mathbf{E}[N] = \frac{7}{8}m$, where m is the number of clauses.

(b) Prove that $\mathbf{P}\left\{N \leq \frac{3m}{4}\right\} \leq \frac{1}{2}$. [Hint: The Markov, Chebyshev, and Chernoff bounds won't work here. Look for a different inequality from Chapter 18 that goes in the right direction.]

(c) Let δ be an arbitrary small constant, $0 < \delta \ll 1$. How can we revise our algorithm to ensure that $\mathbf{P}\left\{N \leq \frac{3m}{4}\right\} \leq \delta$? [Hint: Your runtime will be a function of δ.]

22.6 **From Monte Carlo to Las Vegas**
[Proposed by David Wajc] Let $\mathcal{P}$ be a decision problem. Imagine that one has two Monte Carlo algorithms for deciding $\mathcal{P}$: Algorithm A is true-biased in that it always returns "true" when the answer is "true," but returns "true" with probability $p > 0$ when the answer is false (that is, it has false positives). Algorithm B is false-biased in that it always returns "false" when the answer is "false," but returns "false" with probability p when the answer is true (that is, it has false negatives). Suppose that the two Monte Carlo algorithms terminate in time n^c on inputs of size n (here c is a constant). Your goal is to create a Las Vegas algorithm, $\mathcal{L}$, whose expected runtime is polynomial. What is your Las Vegas algorithm, $\mathcal{L}$, and what is its expected runtime?

22.7 **Approximating π**
In this exercise, you will devise and analyze a Monte Carlo randomized algorithm to approximate π.

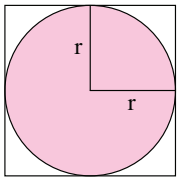


Figure 22.3 *For Exercise 22.7. A circle of radius r embedded within a square.*

(a) Suppose that you throw n darts uniformly at random within the square of Figure 22.3. Let X denote the number of darts which land within the circle. How is X distributed?
(b) Define a random variable Z (related to X) where Z is your "estimator of π." Prove that $\mathbf{E}[Z] = \pi$.
(c) We want to say that, with probability $1 - \epsilon$, Z is within δ of π. How high should n be to ensure this? Assume $0 < \delta \ll 1$ and $0 < \epsilon \ll 1$.

22.8 **Number of min-cuts**
For any graph G, with n vertices and m edges, let $S_1, S_2, \ldots, S_k$ denote the min-cuts of G. We want to prove an upper bound on the value of k.
(a) Let A_i denote the event that S_i is output by the Randomized Min-Cut algorithm. We proved a lower bound on $\mathbf{P}\{A_i\}$. What was that?
(b) Now consider the event: $[A_1 \cup A_2 \cup \cdots \cup A_k]$. By definition $\mathbf{P}\{A_1 \cup A_2 \cup \cdots \cup A_k\} \leq 1$. Use what you learned in part (a) to prove an upper bound on k.

22.9 **Randomized Max-Cut**
You are given a graph $G = (V, E)$ with n vertices and m edges. You want to divide the vertices into two disjoint sets, X and $\overline{X}$, where $X \cup \overline{X} = V$, so as to *maximize* the number of edges between X and $\overline{X}$. Rather than trying all ways of splitting V into two sets, you use this algorithm:

Algorithm 22.13 (Randomized Max-Cut algorithm)
Input: $G = (V, E)$.
Output: Disjoint sets, X and $\overline{X}$, where $X \cup \overline{X} = V$.

For each vertex $x \in V$, flip a fair coin:
If heads, put x into set X.
If tails, put x into set $\overline{X}$.

(a) What is the probability that a given edge $e = (x, y) \in E$ is in the cut-set? What is the expected size of the cut-set produced?

(b) Let

$$Y_e = \begin{cases} 1 & \text{if edge } e \text{ is in the cut-set} \\ 0 & \text{otherwise} \end{cases}.$$

Which of the following are true (may be more than one)?

(i) The Y_e's are independent.
(ii) The Y_e's pair-wise independent.
(iii) The Y_e's are three-wise independent.
(iv) None of the above.

For anything you claimed to be true, provide a proof. For anything you claimed to be false, provide a counter-example.

(c) Use Chebyshev's inequality to show that with high probability, $\left(\geq 1 - O\left(\frac{1}{m}\right)\right)$, the size of the cut-set exceeds $m/4$.

(d) Why couldn't we use Chernoff bounds in part (c)?

22.10 **Amplification of confidence (boosting)**

We are given a Monte Carlo algorithm, $\mathcal{M}$, which always runs in some fixed time t. Given an input, $\mathcal{M}$ returns the correct answer for that input with probability $\frac{2}{3}$ (assume that there is a unique correct answer for each input). With probability $\frac{1}{3}$, $\mathcal{M}$ makes up a value and returns that. Note that the made-up value might be different each time. You do not know whether the output of $\mathcal{M}$ is correct or not.

(a) Given $\epsilon > 0$, explain how to create a new randomized algorithm, $\mathcal{N}$, which outputs the correct answer with probability at least $1 - \epsilon$.

(b) What is the explicit runtime of $\mathcal{N}$ (in terms of t)?

22.11 **BogoSort**

I have an array A of length n containing distinct integers $a_1, a_2, \ldots, a_n$ in a random order. I decide to sort these using a stupid randomized algorithm:

Algorithm 22.14 (BogoSort($a_1, a_2, \ldots, a_n$))

1. *Check if $a_1, a_2, \ldots, a_n$ are sorted. If so, return the sorted array.*
2. *Randomly permute $a_1, a_2, \ldots, a_n$. Then return to step 1.*

Determine the expected number of comparisons needed for BogoSort:

(a) Let C be the number of comparisons needed for line 1 (the check step). What is $\mathbf{E}[C]$? For large n, can you approximate this within 1?

(b) What is the expected number of iterations needed in BogoSort?

(c) What is the expected total number of comparisons needed for BogoSort? (Note: Step 2 doesn't need comparisons.)

22.12 **Monte Carlo control groups**

In running an experiment, it is often useful to have two "equal" sets of people, where we run the experiment on one set, and the other set is used

as a control group. However, dividing people into two "equal" sets is non-trivial. Consider the example shown in the table below: We have m men who are rated on a 0/1 scale in terms of n features. We would like to divide the m men into two groups so that each group is approximately equal with respect to each feature.

	Looks	Brains	Personality
Mike	1	1	0
Alan	0	0	1
Richard	1	0	1
Bill	1	1	0

(a) Intuitively, which grouping is better in terms of creating balance between the groups across each feature:
(i) Group 1: Alan with Richard Group 2: Mike with Bill
(ii) Group 1: Mike with Alan Group 2: Richard with Bill

(b) Mathematically, we express the above table by an $m \times n$ matrix $\mathbf{A}$, where

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

Consider an m-dimensional row vector $\vec{b} = (b_1, b_2, \ldots, b_m)$, where $b_i \in \{1, -1\}$. Let

$$\vec{b} \cdot \mathbf{A} = \vec{c}.$$

Here, the 1's in $\vec{b}$ indicate the men in group 1 and the -1's in $\vec{b}$ indicate the men in group 2. Observe that $\vec{c}$ is an n-dimensional row vector, where c_i indicates the total score for group 1 on feature i minus the total score for group 2 on feature i. Using the above notation, we rephrase our problem as: How can we find a $\vec{b}$ which minimizes $\max_j |c_j|$? Suppose we choose $\vec{b}$ randomly: we set $b_i = 1$ with probability $\frac{1}{2}$ and set $b_i = -1$ with probability $\frac{1}{2}$. Prove

$$\mathbf{P}\left\{\max_j |c_j| \geq \sqrt{4m \ln n}\right\} \leq \frac{2}{n}.$$

[Hint 1: Start by deriving $\mathbf{P}\left\{|c_j| \geq \sqrt{4m \ln n}\right\}$.]
[Hint 2: You will want to use the result in Exercise 18.15.]
[Hint 3: Don't worry about the fact that c_j might have fewer than m summands. Your bound for the case of m summands will still work.]

22.13 **Fixed-length path**

Given a graph G with n vertices, you want to decide whether there exists a path in G containing ℓ *distinct* vertices. The naive brute force algorithm has runtime $O(n^\ell)$. Consider instead this Monte Carlo algorithm:

1. Label each vertex independently and uniformly at random from $\{1, \ldots, \ell\}$.
2. Using breadth-first search (or something similar), check if there is a path $(v_1, \ldots, v_\ell)$ such that for all $1 \leq i \leq \ell$, v_i has label i. If yes, return true. Otherwise return false.

You may assume that this Monte Carlo algorithm runs in time $O(n^2)$.

(a) Show that this algorithm has a one-sided error of at most $1 - \frac{1}{\ell^\ell}$.

(b) How can we lower the error probability to $\frac{1}{n}$? What is the runtime of your new algorithm? [Hint: Recall from (1.12) that $1 + x \leq e^x$, $\forall x \geq 0$.]

22.14 **Generating a random permutation**

We are given an array A containing distinct integers $a_1, a_2, \ldots a_n$. We want to perfectly shuffle the array. That is, every one of the $n!$ permutations should be equally likely to be the result of our shuffle. Below are two potential algorithms. For each algorithm either prove that it results in a perfect shuffle, or provide a counter-example. [Hints: Induction is useful in proofs. Counter-examples shouldn't need more than three integers.]

Algorithm 22.15 (Shuffle attempt 1)

for $i = n, n-1, \ldots, 2$ **do**
    j = *random integer with* $1 \leq j \leq i$
    exchange $A[j]$ *and* $A[i]$
return A

Algorithm 22.16 (Shuffle attempt 2)

for $i = n, n-1, \ldots, 2$ **do**
    j = *random integer with* $1 \leq j \leq n$
    exchange $A[j]$ *and* $A[i]$
return A

22.15 **Approximate median**

In Chapter 21 we saw how to find the exact median of an unsorted list, $L = \{a_1, a_2, \ldots, a_n\}$ of n elements in expected $O(n)$ comparisons using the Median-Select algorithm. Now we present an algorithm to find an "ϵ-approximation of the median" in only expected $O\left(\frac{\log n}{\epsilon^2}\right)$ comparisons. Specifically, if $L' = \{s_1, s_2, \ldots, s_n\}$ denotes a sorted version of L, where $s_{(n+1)/2}$ is the exact median (assume n is odd), then our algorithm returns

an element of L' in the sublist $S = \{s_{\text{low}}, \ldots, s_{\text{high}}\}$, where $low = (\frac{1}{2} - \epsilon)n$ and $high = (\frac{1}{2} + \epsilon)n$. See Figure 22.4. For simplicity, assume that low, $high$, and $(n+1)/2$ are all integers.

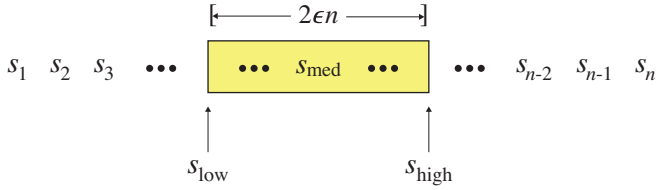


Figure 22.4 *For Exercise 22.15. The approximate median is anything in the box.*

Our approximate-median algorithm works as follows: We select a random element from L t times (the same element might get picked more than once). Let M denote the set of the chosen t elements. Now perform Median-Select on M and return its median, m, as our approximate-median (assume t is odd). You will need to find a t that is sufficiently high for our approximate-median, m, to be within the sublist $S = \{s_{\text{low}}, \ldots, s_{\text{high}}\}$ with high probability.

22.16 **Knockout tournament**

In a knockout tournament, the goal is to determine the "best" of n sports teams. Teams are paired against each other according to the tree structure shown in Figure 22.5. Each *round* starts with some number of "remaining teams." These remaining teams are paired up according to the tree structure. Each pair of teams (A,B) plays k games, where the team with the majority of the wins (say, A), moves up to the next round, and the other team (B) is dropped. Assume ties are impossible and k is odd. Assume also that $n = 2^m$ for some positive integer m. In Figure 22.5, we see that the number of teams starts at $n = 8$, but after one round we're down to four remaining teams, and after the second round, we're down to two teams, and so on.

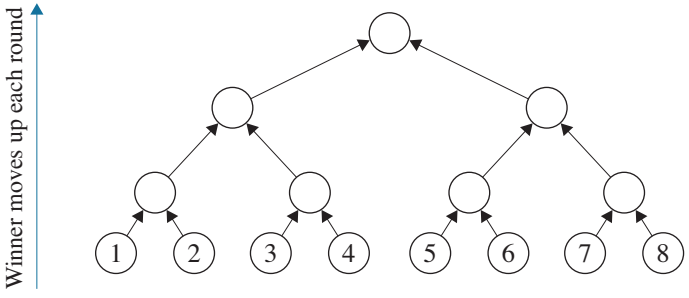


Figure 22.5 *Illustration of knockout tournament with $n = 8$ teams and $m = 3$ rounds.*

Assume that there is a "best" team, b, such that if b is paired against any team j, b will beat j with probability $\geq \frac{1}{2}+\epsilon$, where $0 < \epsilon \leq \frac{1}{2}$. Prove that when $k \geq \frac{4m}{\epsilon^2}$, team b wins the tournament with high probability $(1 - \frac{1}{n})$.
[Hint 1: It will be convenient to use the result from Exercise 18.21(b).]
[Hint 2: The following fact will simplify the algebra: Let N be an integer random variable and k be odd. Then $\mathbf{P}\left\{N \leq \frac{k-1}{2}\right\} = \mathbf{P}\left\{N \leq \frac{k}{2}\right\}$.]
[Hint 3: You will want to show that the probability that team b loses a single round is upper-bounded by e^{-2m}.]
There is a rich literature on applying Chernoff bounds to tournament design (e.g., [1, 2, 78]). This problem is based on [2].

22.17 **Adding n-bit numbers – average-case analysis**

Mor is implementing a bitwise adder to add two *random* n-bit binary numbers $a_1 a_2 \ldots a_n$ and $b_1 b_2 \ldots b_n$ for her operating system. She notices that the conventional adder needs to traverse all n bits of the two numbers from the lowest bit to the highest bit, propagating carry-ins when necessary (when we add a 1 to a 1, we need to carry-in a 1 to the next highest bit). To add faster, Mor constructs the following Near Adder whose work can be parallelized: instead of adding all n-bits sequentially, the Near Adder divides them into $\frac{n}{d}$ segments of size d-bits each. The Near Adder then uses the conventional adder to add consecutive overlapping $2d$-bit "blocks" in *parallel*, as shown in Figure 22.6, where the carry-in to each block is assumed to be 0. Thus the runtime of the Near Adder is $O(2d)$ rather than $O(n)$.

Observe that every (blue) $2d$-bit block (other than the rightmost one) has a shaded (pink) part for the least-significant d bits, and an unshaded part for the most-significant d bits. The Near Adder returns only the most significant d bits (the unshaded part) of each block as the sum of the two n-bit numbers. Notice that the $2d$-bit blocks purposely overlap. Only the most significant d bits of each $2d$-bit computation are returned, because they are likely to be uncorrupted. The pink parts are likely wrong because of the assumed 0 carry-in.

(a) Does the Near Adder always output the correct final sum? If not, when does it fail?

(b) Define a **propagate pair** to be a pair of bits (a_i, b_i) such that either $a_i = 1$ and $b_i = 0$, or $a_i = 0$ and $b_i = 1$. Prove Claim 22.17:

> **Claim 22.17** *The Near Adder is incorrect if and only if the true carry-in to a* $2d$*-bit block is* 1 and *all the lower* d *pairs of bits in that block are propagate pairs.*

(c) Say we want the Near Adder to output the correct sum with probability $1-\epsilon$. How large should we pick d to be? Does picking a large d increase or decrease our accuracy? Here are some steps to follow:

(i) First prove that the probability that the carry-in bit to a $2d$-bit block is 1 is strictly less than $\frac{1}{2}$.

(ii) Now use step (i) to derive an upper bound on the probability that an arbitrary $2d$-bit block causes an error. Use Claim 22.17.

(iii) Apply a union bound over all blocks to determine the probability of error of the Near Adder as a function of d and n.

(iv) What value of d suffices to ensure that the Near Adder provides the correct answer with probability $> 1 - \epsilon$?

(v) Does picking a larger d decrease or increase the error?

(d) (Optional!) Peter wants an Adder which is 100% accurate. Mor proposes that Peter can build upon her Near Adder to achieve 100% accuracy with low *expected* runtime. Propose an algorithm that could achieve this. You do not have to provide details or analysis, just a general idea.

This problem is based on a collaboration between Peter and Mor in [31].

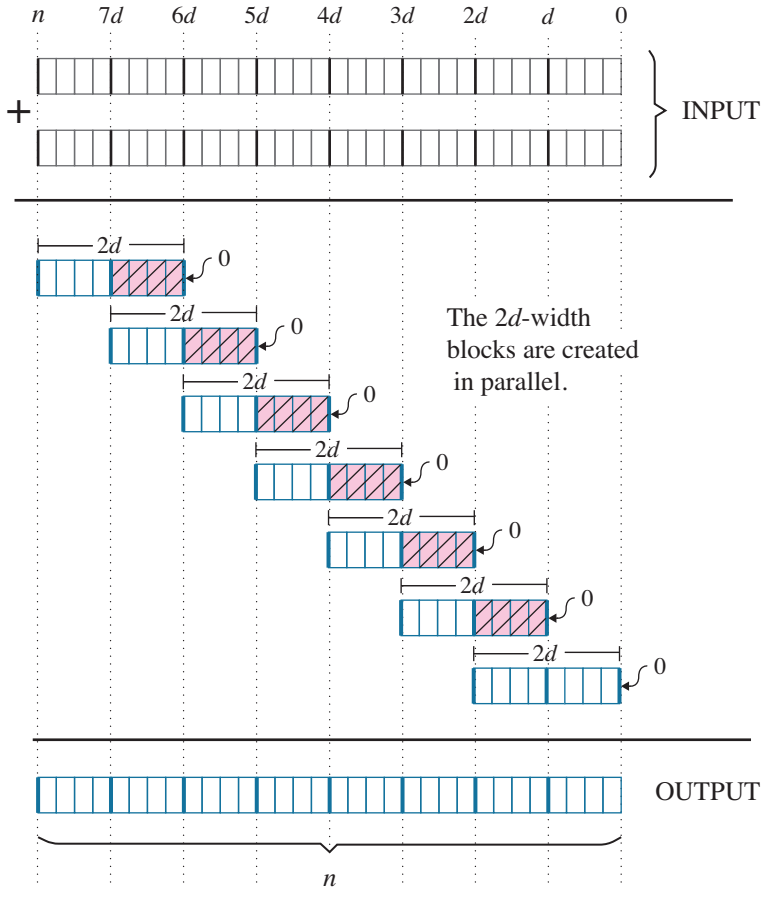


Figure 22.6 *Picture of Near Adder from [31].*