1 Introduction to Randomized Algorithms

1.1 An Introductory Example

This is one of the first published uses of randomization, and is due to Freivalds. Suppose we have an untrustworthy oracle for multiplying matrices, and we want to check whether the oracle was correct on a certain input.

See [Fre79].

Given three $n \times n$ matrices A, B, and C, determine whether $AB = C$. The naive approach of explicit multiplication requires $O(n^3)$ time, or $O(n^{2.376})$ time using the (essentially) best-known algorithm. Freivalds' algorithm provides a probabilistic solution in $O(n^2)$ time with bounded error probability. The idea is to pick a random n-dimensional vector, multiply both sides by it, and see if they're equal. Formally:

1. pick a vector $r = (r_1, \ldots, r_n)$ such that each r_i is i.i.d. uniform from a finite set S with $|S| \geq 2$
2. if $(AB)r = Cr$ then output *no* else output *yes* [one-sided error]

This algorithm runs in $O(n^2)$ time because matrix multiplication is associative, so $(AB)r$ can be computed as $A(Br)$, thus requiring only three matrix-vector multiplications for the algorithm. The algorithm also has one-sided error. If we find an r such that $(AB)r \neq Cr$, then certainly the answer is no. If it outputs yes, we could have been unlucky, but we now show that this happens with probability $\leq 1/2$.

Example 1.1.
Consider matrices:

$$A = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix}, \; B = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}, \; C = \begin{pmatrix} 6 & 5 \\ 8 & 7 \end{pmatrix}$$

If we choose $r = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we can find $ABr = \begin{pmatrix} 2 & 3 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \neq Cr$. Clearly it follows that $AB \neq C$. However, we may choose $r = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, which gives $ABr = Cr$.

Proposition 1.1.
If $AB \neq C$, then $\Pr[ABr = Cr] \leq \frac{1}{|S|}$.

Proof. If $AB \neq C$, let $D = AB - C \neq 0$. For a random r, we would like to show that

$$\Pr[Dr = 0] \leq \frac{1}{|S|}.$$

This follows because D has at least one non-zero entry, without loss of generality, say $D_{n,n}$. Then $(Dr)_n = 0$ gives a linear equation

$$r_n = -\frac{1}{D_{n,n}} \cdot (D_{1,1} r_1 + \cdots + D_{n,n-1} r_{n-1}),$$

which has at most one solution r_n in S for each choice of the values $r_1, \ldots, r_n$. □

This analysis illustrates the simple but powerful "principle of deferred decisions": we first fixed the choices for $r_1, \ldots, r_{n-1}$, and then considered the effect of the random choice of r_n given those choices.

There are at least two ways to further decrease the error probability above. We can enlarge the set S. But a clever idea is to independently run the algorithm many times. Suppose we run the algorithm t times independently. If we get *no* at least once, output *no*; otherwise, output *yes*. Since each trial is independent, $\Pr[\text{error}] \leq |S|^{-t} \leq 2^{-t}$. The whole algorithm still has run-time $O(n^2)$.

Algorithm 1 Freivalds' Matrix Verification

Require: $A, B, C \in \mathbb{R}^{n \times n}$
Ensure: **True** if $AB = C$, **False** otherwise (with probability $\geq 1 - 2^{-t}$)

```
1: for k = 1 to t do
2:     Generate random binary vector r ∈ {0,1}^n
3:     Compute x = Br, y = Ax, z = Cr
4:     if y ≠ z then
5:         return False
6:     end if
7: end for
8: return True
```

1.2 Polynomial identity testing

A similar example is the problem of polynomial identity testing. We are given two polynomials $F(x), G(x) \in \mathbb{C}[x]$. The problem is to decide whether $F(x) \equiv G(x)$.

When formally defining a computational problem, one needs to explicitly specify how the input is encoded. Some practical encoding methods are:

1. A polynomial is represented as products of small polynomials, e.g., $F(x) = (x-1)(x-2)(x+3)$.
2. A polynomial is given by their coefficients, e.g., $F(x) = f_0 + f_1 x + \cdots + f_d x^d$.
3. A polynomial is treated as an *evaluation oracle*, where given $s \in \mathbb{C}$, a black box returns $F(s)$.

If both polynomials are represented by coefficients, namely, $F(x) = \sum_{i=0}^{d} f_i x^i$ and $G(x) = \sum_{i=0}^{d} g_i x^i$, then the identity testing is a trivial task as one scanning of all coefficients suffices.

Assuming $F(x) = \prod_{i=1}^{d} (x - a_i)$ and $G(x) = \sum_{i=0}^{d} b_i x^i$, a basic strategy is to expand $F(x)$ and compute all the coefficients. If we assume that the multiplication and addition of two elements in $\mathbb{C}$ cost constant time, a straightforward implementation requires $O(d^2)$ time. One can use Fast Fourier Transform to accelerate the polynomial multiplication, so that the overall running time can be reduced to $O(d \log d)$.

However, if random bits are allowed to use, we can decide whether $F(x) \equiv G(x)$ in linear time, at the cost of making mistakes with very low probability, say 0.01%. Again, we use the simple "witness" or "fingerprint" idea: randomly pick a number $x \in U \subseteq \mathbb{C}$ and check if $F(x) = G(x)$.

It is clear that if $F(x) \equiv G(x)$, our algorithm always outputs the correct answer. On the other hand, if $F(x) \not\equiv G(x)$, the algorithm might make mistakes if r is a root of the polynomial $F(x)-G(x)$. To analyze the chance of this event, we need the *Fundamental Theorem of Algebra*.

Theorem 1.2. (Fundamental Theorem of Algebra)
Every non-zero polynomial $F(x) \in \mathbb{C}[x]$ of degree d has, counted with multiplicity, exactly d roots in $\mathbb{C}$.

For $|U| \geq 100d$:

$$\Pr_{r \in_R U}[F(r)-G(r)=0] \leq \frac{d}{|U|} \leq \frac{1}{100}$$

Again, to reduce the error probability, we can repeat t-times to reduce error to 100^{-t}.

Actually, the benefit of this simple randomized algorithm turns out to be more prominent in the multivariate case than the univariate case.

Suppose $F, G \in \mathbb{C}[x_1, \ldots, x_n]$. Now the naive approach to expand F and G as sums of monomials and compare their coefficients does not work, since the number of monomials may be exponential in the size of input, e.g., $\prod_{i=1}^{n-1}(x_i + x_{i+1})$. In fact, there is no known polynomial time deterministic algorithm for this problem. On the other hand, how about using randomness? We still fix a set $U \subseteq \mathbb{C}$, independently sample n random numbers $r_1, \ldots, r_n \in U$ uniformly and check

We may assume that the given polynomials can be evaluated efficiently at any given point $(x_1, \ldots, x_n)$.

$$F(r_1, \ldots, r_n) \stackrel{?}{=} G(r_1, \ldots, r_n).$$

The remaining part is to bound the error probability, namely,

$$\Pr[F(r_1, \ldots, r_n) \neq G(r_1, \ldots, r_n) \mid F \not\equiv G].$$

Theorem 1.3. Schwartz-Zippel Theorem
Let $Q \in \mathbb{C}[x_1, \ldots, x_n]$ be a non-zero multivariate polynomial of degree at most d. For any set $U \subseteq \mathbb{C}$:

$$\Pr_{r_1, \ldots, r_n \in_R U}[Q(r_1, \ldots, r_n) = 0] \leq \frac{d}{|U|}.$$

Proof. We prove the theorem by induction on n, the number of variables.

Base Case ($n=1$): Follows directly from the Fundamental Theorem of Algebra. A degree d univariate polynomial has at most d roots, so:

$$\Pr[Q(r_1) = 0] \leq \frac{d}{|U|}$$

Inductive Step: Assume the theorem holds for all polynomials with $(n-1)$ variables. Let Q be a polynomial in n variables with degree d.

Express Q as:

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_n^i \cdot Q_i(x_1, \ldots, x_{n-1})$$

where:

- $Q_k \neq 0$ (non-zero leading coefficient)
- $\deg(Q_k) \leq d-k$

Let $\mathcal{E}$ be the event $Q(r_1, \ldots, r_n) = 0$. We analyze the probability using two cases:

1. $Q_k(r_1, \ldots, r_{n-1}) = 0$:
   By induction hypothesis:

$$\Pr[Q_k(r_1, \ldots, r_{n-1}) = 0] \leq \frac{d-k}{|U|}$$

2. $Q_k(r_1, \ldots, r_{n-1}) \neq 0$:
   Fix $r_1, \ldots, r_{n-1}$. Now Q becomes a univariate polynomial in x_n:

$$P_{r_1,\ldots,r_{n-1}}(x_n) = \sum_{i=0}^{k} x_n^i \cdot Q_i(r_1, \ldots, r_{n-1})$$

   This polynomial has degree $\leq k$. By base case:

$$\Pr[P(\cdot) = 0 \mid Q_k \neq 0] \leq \frac{k}{|U|}$$

Using the law of total probability:

$$\Pr[\mathcal{E}] \leq \underbrace{\Pr[Q_k = 0]}_{\leq \frac{d-k}{|U|}} + \underbrace{\Pr[Q_k \neq 0]}_{\leq 1} \cdot \underbrace{\Pr[\mathcal{E} \mid Q_k \neq 0]}_{\leq \frac{k}{|U|}} \leq \frac{d-k}{|U|} + \frac{k}{|U|} = \frac{d}{|U|}$$

Thus by induction, the theorem holds for all $n \geq 1$. □

Thus, if we take the set U to have cardinality at least twice the degree of our polynomials, we can bound the probability of error by $1/2$. This can be reduced to any desired small number by repeated trials, as usual.

1.3 Karger's Min-Cut Algorithm

The **min-cut problem** is a fundamental problem in graph theory. Given an undirected graph $G = (V, E)$, a **cut** is a set of edges $C \subseteq E$ whose removal disconnects the graph (or equivalently, the edges connecting S and $V \setminus S$ for some $S \subseteq V$). The goal of the min-cut problem is to find the cut with the smallest cardinality.

You might have learnt that the problem can be solved in polynomial-time using a max-flow based algorithm: Fix a source s and enumerate all possible sinks t. For each pair of the source and the sink (s,t), one can compute the max-flow between s and t. Let S be the set of vertices connected to s after removing all edges with flow. Then a min-cut between s and t is the set of edges connecting S and its complement, denoted by $E(S, \bar{S})$. The global min-cut is therefore the minimum one over all t's. The famous max-flow min-cut theorem guarantees the correctness of the algorithm. The running time depends on the algorithm to compute max-flow between s and t. With the fastest algorithm so far [Che+22], the total cost is $O(nm^{1+o(1)})$ where $n = |V|$ and $m = |E|$.

Karger [Kar93] presented a very simple randomized algorithm for the problem. The main operation used in the algorithm is the *contraction* of edges. Given a graph $G = (V, E)$ and an edge $e = \{u, v\} \in E$, the contraction of e merges u and v and removes all edges between u and v. Karger's algorithm works by repeatedly contracting edges until only two vertices remain. The edges between these two vertices represent a cut.

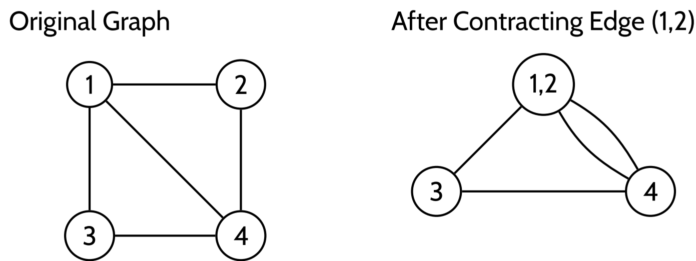

Figure 1.1: Example of edge contraction in a graph. The edge between vertices 1 and 2 is contracted, merging them into a single vertex.

Algorithm 2 Karger's Min-Cut Algorithm

1: **Input:** An undirected graph $G = (V, E)$
2: **Output:** A cut of G
3: **while** $|V| > 2$ **do**
4: Randomly select an edge $e \in E$
5: Contract the edge e, merging the two vertices it connects
6: Remove all self-loops
7: **end while**
8: **Return:** The cut represented by the remaining edges

Example 1.2.
Consider the same graph as before. Suppose we randomly select the edge $(1, 2)$ to contract. After contraction, vertices 1 and 2 are merged into a single vertex, say $\{1, 2\}$. The graph now has 3 vertices: $\{1, 2\}$, 3, and 4. The edges are updated accordingly, and any self-loops are removed. Suppose we randomly contract the edge $(\{1, 2\}, 3)$ next. Then we finally obtain a graph consisting of 2 vertices: $\{1, 2, 3\}$ and 4, and 3 edges between them. Thus the algorithm terminates and outputs 3.

Karger's algorithm is a Monte Carlo algorithm, meaning it may return an incorrect result with some probability. The key question is: what is the probability that the algorithm returns the correct min-cut?

We aim to prove that the probability of Karger's algorithm returning the minimum cut is at least $\frac{2}{n(n-1)}$, where n is the number of vertices in the graph.

Let k be the size of the minimum cut in G. There are some key observations: at each step of the algorithm,

1. the minimum cut in current graph has at least k edges;
2. the degree of each vertex in the current G is at least k (since otherwise, removing the edges incident to that vertex would yield a smaller cut);
3. the total number of edges in the current G is $\geq \frac{n'k}{2}$ (where n' is the number of vertices in current graph).

At each step of the algorithm, we contract a randomly chosen edge. The algorithm succeeds if none of the edges in the minimum cut C are contracted during the process. We now bound the probability of this event. Let E_i be the event that an edge from C is contracted in the i-th step. The probability of success is the probability that none of the E_i events occur.

At the beginning, the graph has n vertices and m edges. The probability of contracting an edge from C in the first step is:

$$\Pr[E_1] = \frac{k}{m} \le \frac{k}{\frac{nk}{2}} = \frac{2}{n}\,.$$

After the first contraction, the graph has $n-1$ vertices and at least $\frac{(n-1)k}{2}$ edges. The probability of contracting an edge from C in the second step, given that no edge from C was contracted in the first step, is:

$$\Pr[E_2 \mid \overline{E_1}] \le \frac{k}{\frac{(n-1)k}{2}} = \frac{2}{n-1}\,.$$

Similarly, after i contractions, the graph has $n-i$ vertices and at least $\frac{(n-i)k}{2}$ edges. The probability of contracting an edge from C in the $(i+1)$-th step, given that no edge from C was contracted in the first i steps, is:

$$\Pr[E_{i+1} \mid \overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_i}] \le \frac{k}{\frac{(n-i)k}{2}} = \frac{2}{n-i}\,.$$

The probability that none of the edges in C are contracted during the entire process is:

$$\begin{aligned}
\Pr[\text{success}] &= \Pr[\overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-2}}] \\
&= \Pr[\overline{E_1}]\Pr[\overline{E_2} \mid \overline{E_1}] \cdots \Pr[\overline{E_{n-2}} \mid \overline{E_1} \cap \overline{E_2} \cap \cdots \cap \overline{E_{n-3}}] \\
&\ge \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{1}{3} \\
&= \frac{2}{n(n-1)}\,.
\end{aligned}$$

Thus, the probability that Karger's algorithm returns the minimum cut is at least $\frac{2}{n(n-1)}$. To increase the probability of success, the algorithm can be run multiple times. If the algorithm is run t times, the probability that at least one run returns the correct min-cut is:

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^t$$

For example, if $t = 100n^2$, the probability becomes

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^{100n^2} \ge 1 - \mathrm{e}^{-200}\,,$$

which is quite close to 1. If $t = n^2 \ln n$, the probability of success approaches 1 as n grows.

What is the time cost of this randomized algorithm? If we maintain the contraction by the union-find set, the cost can be reduced to $\tilde{O}(n^2 m)$.

The notation $\tilde{O}$ is the big-O notation that ignores logarithmic factors. Namely, $f(n) \in \tilde{O}(T)$ is equivalent to:

$$\exists k \in \mathbb{N}, f(n) \in O(T \log^k n)\,.$$

References

[Che+22] Li Chen et al. "Maximum Flow and Minimum-Cost Flow in Almost-Linear Time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 612–623. DOI: 10.1109/FOCS54457.2022.00064.

[Fre79] Rūsiņš Freivalds. “Probabilistic Machines Can Use Less Running Time”. In: *IFIP Congress*. One of the earliest formal treatments of probabilistic computation. 1979, pp. 839–842.

[Kar93] David R. Karger. “Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm”. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’93. Austin, Texas, USA: Society for Industrial and Applied Mathematics, 1993, pp. 21–30. ISBN: 0898713137.