Verifying Polynomial Identities

Computers can sometimes make mistakes, due for example to incorrect programming or hardware failure. It would be useful to have simple ways to double-check the results of computations. For some problems, we can use randomness to efficiently verify the correctness of an output.

Suppose we have a program that multiplies together monomials. Consider the problem of verifying the following identity, which might be output by our program:

$$(x+1)(x-2)(x+3)(x-4)(x+5)(x-6) \stackrel{?}{\equiv} x^6 - 7x^3 + 25.$$

There is an easy way to verify whether the identity is correct: multiply together the terms on the left-hand side and see if the resulting polynomial matches the right-hand side. In this example, when we multiply all the constant terms on the left, the result does not match the constant term on the right, so the identity cannot be valid. More generally, given two polynomials $F(x)$ and $G(x)$, we can verify the identity

$$F(x) \stackrel{?}{\equiv} G(x)$$

by converting the two polynomials to their canonical forms $\left(\sum_{i=0}^{d} c_i x^i\right)$; two polynomials are equivalent if and only if all the coefficients in their canonical forms are equal. From this point on let us assume that, as in our example, $F(x)$ is given as a product $F(x) = \prod_{i=1}^{d}(x - a_i)$ and $G(x)$ is given in its canonical form. Transforming $F(x)$ to its canonical form by consecutively multiplying the ith monomial with the product of the first $i-1$ monomials requires $\Theta(d^2)$ multiplications of coefficients. We assume in what follows that each multiplication can be performed in constant time, although if the products of the coefficients grow large then it could conceivably require more than constant time to add and multiply numbers together.

So far, we have not said anything particularly interesting. To check whether the computer program has multiplied monomials together correctly, we have suggested multiplying the monomials together again to check the result. Our approach for checking the program is to write another program that does essentially the same thing we expect the first program to do. This is certainly one way to double-check a program: write a second program that does the same thing, and make sure they agree. There are at least two problems with this approach, both stemming from the idea that there should be a difference between checking a given answer and recomputing it. First, if there is a bug in the program that multiplies monomials, the same bug may occur in the checking program. (Suppose that the checking program was written by the same person who wrote the original program!) Second, it stands to reason that we would like to check the answer in less time than it takes to try to solve the original problem all over again.

Let us instead utilize randomness to obtain a faster method to verify the identity. We informally explain the algorithm and then set up the formal mathematical framework for analyzing the algorithm.

Assume that the maximum degree, or the largest exponent of x, in $F(x)$ and $G(x)$ is d. The algorithm chooses an integer r uniformly at random in the range $\{1, \dots, 100d\}$, where by "uniformly at random" we mean that all integers are equally likely to be chosen. The algorithm then computes the values $F(r)$ and $G(r)$. If $F(r) \neq G(r)$ the algorithm decides that the two polynomials are not equivalent, and if $F(r) = G(r)$ the algorithm decides that the two polynomials are equivalent.

Suppose that in one computation step the algorithm can generate an integer chosen uniformly at random in the range $\{1, \dots, 100d\}$. Computing the values of $F(r)$ and $G(r)$ can be done in $O(d)$ time, which is faster than computing the canonical form of $F(r)$. The randomized algorithm, however, may give a wrong answer.

How can the algorithm give the wrong answer?

If $F(x) \equiv G(x)$, then the algorithm gives the correct answer, since it will find that $F(r) = G(r)$ for any value of r.

If $F(x) \not\equiv G(x)$ and $F(r) \neq G(r)$, then the algorithm gives the correct answer since it has found a case where $F(x)$ and $G(x)$ disagree. Thus, when the algorithm decides that the two polynomials are not the same, the answer is always correct.

If $F(x) \not\equiv G(x)$ and $F(r) = G(r)$, the algorithm gives the wrong answer. In other words, it is possible that the algorithm decides that the two polynomials are the same when they are not. For this error to occur, r must be a root of the equation $F(x) - G(x) = 0$. The degree of the polynomial $F(x) - G(x)$ is no larger than d and, by the fundamental theorem of algebra, a polynomial of degree up to d has no more than d roots. Thus, if $F(x) \not\equiv G(x)$, then there are no more than d values in the range $\{1, \dots, 100d\}$ for which $F(r) = G(r)$. Since there are $100d$ values in the range $\{1, \dots, 100d\}$, the chance that the algorithm chooses such a value and returns a wrong answer is no more than $1/100$.

Verifying Matrix Multiplication

We now consider another example where randomness can be used to verify an equality more quickly than the known deterministic algorithms. Suppose we are given three $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. For convenience, assume we are working over the integers modulo 2. We want to verify whether

$$\mathbf{AB} = \mathbf{C}.$$

One way to accomplish this is to multiply $\mathbf{A}$ and $\mathbf{B}$ and compare the result to $\mathbf{C}$. The simple matrix multiplication algorithm takes $\Theta(n^3)$ operations. There exist more sophisticated algorithms that are known to take roughly $\Theta(n^{2.37})$ operations.

Once again, we use a randomized algorithm that allows for faster verification – at the expense of possibly returning a wrong answer with small probability. The algorithm is similar in spirit to our randomized algorithm for checking polynomial identities. The algorithm chooses a random vector $\bar{r} = (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$. It then computes $\mathbf{AB}\bar{r}$ by first computing $\mathbf{B}\bar{r}$ and then $\mathbf{A}(\mathbf{B}\bar{r})$, and it also computes $\mathbf{C}\bar{r}$. If $\mathbf{A}(\mathbf{B}\bar{r}) \neq \mathbf{C}\bar{r}$, then $\mathbf{AB} \neq \mathbf{C}$. Otherwise, it returns that $\mathbf{AB} = \mathbf{C}$.

The algorithm requires three matrix-vector multiplications, which can be done in time $\Theta(n^2)$ in the obvious way. The probability that the algorithm returns that $\mathbf{AB} = \mathbf{C}$ when they are actually not equal is bounded by the following theorem.

Theorem 1.4: *If* $\mathbf{AB} \neq \mathbf{C}$ *and if* $\bar{r}$ *is chosen uniformly at random from* $\{0,1\}^n$, *then*

$$\Pr(\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}) \leq \frac{1}{2}.$$

Proof: Before beginning, we point out that the sample space for the vector $\bar{r}$ is the set $\{0,1\}^n$ and that the event under consideration is $\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}$. We also make note of the following simple but useful lemma.

Lemma 1.5: *Choosing* $\bar{r} = (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$ *uniformly at random is equivalent to choosing each* r_i *independently and uniformly from* $\{0,1\}$.

Proof: If each r_i is chosen independently and uniformly at random, then each of the 2^n possible vectors $\bar{r}$ is chosen with probability 2^{-n}, giving the lemma. $\square$

Let $\mathbf{D} = \mathbf{AB} - \mathbf{C} \neq 0$. Then $\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}$ implies that $\mathbf{D}\bar{r} = 0$. Since $\mathbf{D} \neq 0$ it must have some nonzero entry; without loss of generality, let that entry be d_{11}.

For $\mathbf{D}\bar{r} = 0$, it must be the case that

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

or, equivalently,

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}. \tag{1.1}$$

Now we introduce a helpful idea. Instead of reasoning about the vector $\bar{r}$, suppose that we choose the r_k independently and uniformly at random from $\{0, 1\}$ in order, from r_n down to r_1. Lemma 1.5 says that choosing the r_k in this way is equivalent to choosing a vector $\bar{r}$ uniformly at random. Now consider the situation just before r_1 is chosen. At this point, the right-hand side of Eqn. (1.1) is determined, and there is at most one choice for r_1 that will make that equality hold. Since there are two choices for r_1, the equality holds with probability at most $1/2$, and hence the probability that $\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}$ is at most $1/2$. By considering all variables besides r_1 as having been set, we have reduced the sample space to the set of two values $\{0, 1\}$ for r_1 and have changed the event being considered to whether Eqn. (1.1) holds.

This idea is called the *principle of deferred decisions*. When there are several random variables, such as the r_i of the vector $\bar{r}$, it often helps to think of some of them as being set at one point in the algorithm with the rest of them being left random – or deferred – until some further point in the analysis. Formally, this corresponds to conditioning on the revealed values; when some of the random variables are revealed, we must condition on the revealed values for the rest of the analysis. We will see further examples of the principle of deferred decisions later in the book.

To formalize this argument, we first introduce a simple fact, known as the law of total probability.

Theorem 1.6 [Law of Total Probability]: *Let $E_1, E_2, \ldots, E_n$ be mutually disjoint events in the sample space Ω, and let $\bigcup_{i=1}^{n} E_i = \Omega$. Then*

$$\Pr(B) = \sum_{i=1}^{n} \Pr(B \cap E_i) = \sum_{i=1}^{n} \Pr(B \mid E_i) \Pr(E_i).$$

Proof: Since the events $B \cap E_i$ $(i = 1, \ldots, n)$ are disjoint and cover the entire sample space Ω, it follows that

$$\Pr(B) = \sum_{i=1}^{n} \Pr(B \cap E_i).$$

Further,

$$\sum_{i=1}^{n} \Pr(B \cap E_i) = \sum_{i=1}^{n} \Pr(B \mid E_i) \Pr(E_i)$$

by the definition of conditional probability. □

Now, using this law and summing over all collections of values $(x_2, x_3, x_4, \dots, x_n) \in \{0,1\}^{n-1}$ yields

$$\begin{aligned}
&\Pr(\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}) \\
&\quad = \sum_{(x_2,\dots,x_n)\in\{0,1\}^{n-1}} \Pr\big((\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}) \cap ((r_2,\dots,r_n) = (x_2,\dots,x_n))\big) \\
&\quad \le \sum_{(x_2,\dots,x_n)\in\{0,1\}^{n-1}} \Pr\left(\left(r_1 = -\frac{\sum_{j=2}^{n} d_{1j}r_j}{d_{11}}\right) \cap ((r_2,\dots,r_n) = (x_2,\dots,x_n))\right) \\
&\quad = \sum_{(x_2,\dots,x_n)\in\{0,1\}^{n-1}} \Pr\left(r_1 = -\frac{\sum_{j=2}^{n} d_{1j}r_j}{d_{11}}\right) \cdot \Pr((r_2,\dots,r_n) = (x_2,\dots,x_n)) \\
&\quad \le \sum_{(x_2,\dots,x_n)\in\{0,1\}^{n-1}} \frac{1}{2}\Pr((r_2,\dots,r_n) = (x_2,\dots,x_n)) \\
&\quad = \frac{1}{2}.
\end{aligned}$$

Here we have used the independence of r_1 and $(r_2, \dots, r_n)$ in the fourth line. ■

To improve on the error probability of Theorem 1.4, we can again use the fact that the algorithm has a one-sided error and run the algorithm multiple times. If we ever find an $\bar{r}$ such that $\mathbf{AB}\bar{r} \neq \mathbf{C}\bar{r}$, then the algorithm will correctly return that $\mathbf{AB} \neq \mathbf{C}$. If we always find $\mathbf{AB}\bar{r} = \mathbf{C}\bar{r}$, then the algorithm returns that $\mathbf{AB} = \mathbf{C}$ and there is some probability of a mistake. Choosing $\bar{r}$ with replacement from $\{0,1\}^n$ for each trial, we obtain that, after k trials, the probability of error is at most 2^{-k}. Repeated trials increase the running time to $\Theta(kn^2)$.

Suppose we attempt this verification 100 times. The running time of the randomized checking algorithm is still $\Theta(n^2)$, which is faster than the known deterministic algorithms for matrix multiplication for sufficiently large n. The probability that an incorrect algorithm passes the verification test 100 times is 2^{-100}, an astronomically small number. In practice, the computer is much more likely to crash during the execution of the algorithm than to return a wrong answer.

An interesting related problem is to evaluate the gradual change in our confidence in the correctness of the matrix multiplication as we repeat the randomized test. Toward that end we introduce Bayes' law.

Theorem 1.7 [Bayes' Law]: *Assume that $E_1, E_2, \dots, E_n$ are mutually disjoint sets such that $\bigcup_{i=1}^{n} E_i = E$. Then*

$$\Pr(E_j \mid B) = \frac{\Pr(E_j \cap B)}{\Pr(B)} = \frac{\Pr(B \mid E_j)\Pr(E_j)}{\sum_{i=1}^{n}\Pr(B \mid E_i)\Pr(E_i)}.$$

As a simple application of Bayes' law, consider the following problem. We are given three coins and are told that two of the coins are fair and the third coin is biased, landing heads with probability $2/3$. We are not told which of the three coins is biased. We permute the coins randomly, and then flip each of the coins. The first and second coins come up heads, and the third comes up tails. What is the probability that the first coin is the biased one?

The coins are in a random order and so, before our observing the outcomes of the coin flips, each of the three coins is equally likely to be the biased one. Let E_i be the event that the ith coin flipped is the biased one, and let B be the event that the three coin flips came up heads, heads, and tails.

Before we flip the coins we have $\Pr(E_i) = 1/3$ for all i. We can also compute the probability of the event B conditioned on E_i:

$$\Pr(B \mid E_1) = \Pr(B \mid E_2) = \frac{2}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{6},$$

and

$$\Pr(B \mid E_3) = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{3} = \frac{1}{12}.$$

Applying Bayes' law, we have

$$\Pr(E_1 \mid B) = \frac{\Pr(B \mid E_1)\Pr(E_1)}{\sum_{i=1}^{3}\Pr(B \mid E_i)\Pr(E_i)} = \frac{2}{5}.$$

Thus, the outcome of the three coin flips increases the likelihood that the first coin is the biased one from $1/3$ to $2/5$.

Returning now to our randomized matrix multiplication test, we want to evaluate the increase in confidence in the matrix identity obtained through repeated tests. In the Bayesian approach one starts with a *prior* model, giving some initial value to the model parameters. This model is then modified, by incorporating new observations, to obtain a *posterior* model that captures the new information.

In the matrix multiplication case, if we have no information about the process that generated the identity then a reasonable prior assumption is that the identity is correct with probability $1/2$. If we run the randomized test once and it returns that the matrix identity is correct, how does this change our confidence in the identity?

Let E be the event that the identity is correct, and let B be the event that the test returns that the identity is correct. We start with $\Pr(E) = \Pr(\bar{E}) = 1/2$, and since the test has a one-sided error bounded by $1/2$, we have $\Pr(B \mid E) = 1$ and $\Pr(B \mid \bar{E}) \leq 1/2$. Applying Bayes' law yields

$$\Pr(E \mid B) = \frac{\Pr(B \mid E)\Pr(E)}{\Pr(B \mid E)\Pr(E) + \Pr(B \mid \bar{E})\Pr(\bar{E})} \geq \frac{1/2}{1/2 + 1/2 \cdot 1/2} = \frac{2}{3}.$$

Assume now that we run the randomized test again and it again returns that the identity is correct. After the first test, I may naturally have revised my prior model, so that I believe $\Pr(E) \geq 2/3$ and $\Pr(\bar{E}) \leq 1/3$. Now let B be the event that the new test returns that the identity is correct; since the tests are independent, as before we have $\Pr(B \mid E) = 1$ and $\Pr(B \mid \bar{E}) \leq 1/2$. Applying Bayes' law then yields

$$\Pr(E \mid B) \geq \frac{2/3}{2/3 + 1/3 \cdot 1/2} = \frac{4}{5}.$$

In general: If our prior model (before running the test) is that $\Pr(E) \geq 2^i/(2^i + 1)$ and if the test returns that the identity is correct (event B), then

$$\Pr(E \mid B) \geq \frac{\dfrac{2^i}{2^i+1}}{\dfrac{2^i}{2^i+1} + \dfrac{1}{2}\dfrac{1}{2^i+1}} = \frac{2^{i+1}}{2^{i+1}+1} = 1 - \frac{1}{2^i+1}.$$

Thus, if all 100 calls to the matrix identity test return that the identity is correct, our confidence in the correctness of this identity is at least $1 - 1/(2^{100} + 1)$.

A Randomized Min-Cut Algorithm

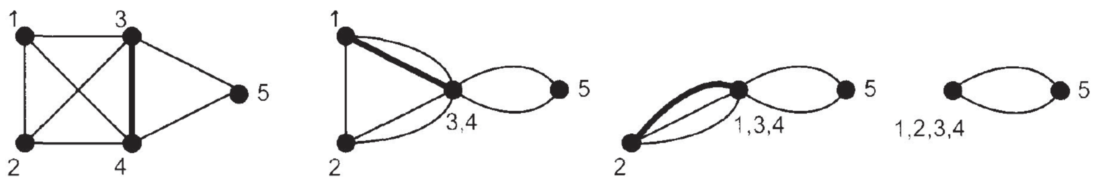
A *cut-set* in a graph is a set of edges whose removal breaks the graph into two or more connected components. Given a graph $G = (V, E)$ with n vertices, the minimum cut – or *min-cut* – problem is to find a minimum cardinality cut-set in G. Minimum cut problems arise in many contexts, including the study of network reliability. In the case where nodes correspond to machines in the network and edges correspond to connections between machines, the min-cut is the smallest number of edges that can fail before some pair of machines cannot communicate. Minimum cuts also arise in clustering problems. For example, if nodes represent Web pages (or any documents in a hypertext-based system) and two nodes have an edge between them if the corresponding nodes have a hyperlink between them, then small cuts divide the graph into clusters of documents with few links between clusters. Documents in different clusters are likely to be unrelated.

We shall proceed by making use of the definitions and techniques presented so far in order to analyze a simple randomized algorithm for the min-cut problem. The main operation in the algorithm is *edge contraction*. In contracting an edge $\{u, v\}$ we merge the two vertices u and v into one vertex, eliminate all edges connecting u and v, and retain all other edges in the graph. The new graph may have parallel edges but no self-loops. Examples appear in Figure 1.1, where in each step the dark edge is being contracted.

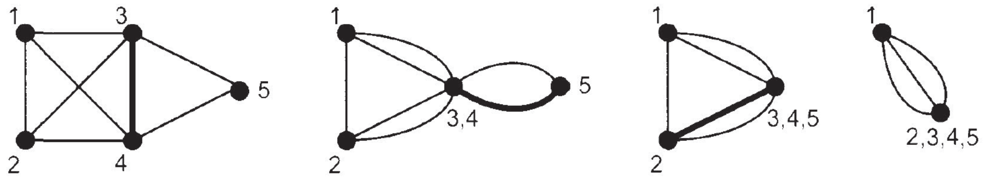
The algorithm consists of $n - 2$ iterations. In each iteration, the algorithm picks an edge from the existing edges in the graph and contracts that edge. There are many possible ways one could choose the edge at each step. Our randomized algorithm chooses the edge uniformly at random from the remaining edges.

Each iteration reduces the number of vertices in the graph by one. After $n - 2$ iterations, the graph consists of two vertices. The algorithm outputs the set of edges connecting the two remaining vertices.

It is easy to verify that any cut-set of a graph in an intermediate iteration of the algorithm is also a cut-set of the original graph. On the other hand, not every cut-set of the original graph is a cut-set of a graph in an intermediate iteration, since some edges of the cut-set may have been contracted in previous iterations. As a result, the output of the algorithm is always a cut-set of the original graph but not necessarily the minimum cardinality cut-set (see Figure 1.1).



(a) A successful run of min-cut.



(b) An unsuccessful run of min-cut.

Figure 1.1: An example of two executions of min-cut in a graph with minimum cut-set of size 2.

We now establish a lower bound on the probability that the algorithm returns a correct output.

Theorem 1.8: *The algorithm outputs a min-cut set with probability at least* $2/n(n-1)$.

Proof: Let k be the size of the min-cut set of G. The graph may have several cut-sets of minimum size. We compute the probability of finding one specific such set C.

Since C is a cut-set in the graph, removal of the set C partitions the set of vertices into two sets, S and $V - S$, such that there are no edges connecting vertices in S to vertices in $V - S$. Assume that, throughout an execution of the algorithm, we contract only edges that connect two vertices in S or two vertices in $V - S$, but not edges in C. In that case, all the edges eliminated throughout the execution will be edges connecting vertices in S or vertices in $V - S$, and after $n - 2$ iterations the algorithm returns a graph with two vertices connected by the edges in C. We may therefore conclude that, if the algorithm never chooses an edge of C in its $n - 2$ iterations, then the algorithm returns C as the minimum cut-set.

This argument gives some intuition for why we choose the edge at each iteration uniformly at random from the remaining existing edges. If the size of the cut C is small and if the algorithm chooses the edge uniformly at each step, then the probability that the algorithm chooses an edge of C is small – at least when the number of edges remaining is large compared to C.

Let E_i be the event that the edge contracted in iteration i is not in C, and let $F_i = \bigcap_{j=1}^{i} E_j$ be the event that no edge of C was contracted in the first i iterations. We need to compute $\Pr(F_{n-2})$.

We start by computing $\Pr(E_1) = \Pr(F_1)$. Since the minimum cut-set has k edges, all vertices in the graph must have degree k or larger. If each vertex is adjacent to at least k edges, then the graph must have at least $nk/2$ edges. The first contracted edge is chosen uniformly at random from the set of all edges. Since there are at least $nk/2$ edges in the graph and since C has k edges, the probability that we do not choose an edge of C in the first iteration is given by

$$\Pr(E_1) = \Pr(F_1) \geq 1 - \frac{2k}{nk} = 1 - \frac{2}{n}.$$

Let us suppose that the first contraction did not eliminate an edge of C. In other words, we condition on the event F_1. Then, after the first iteration, we are left with an $(n-1)$-node graph with minimum cut-set of size k. Again, the degree of each vertex in the graph must be at least k, and the graph must have at least $k(n-1)/2$ edges. Thus,

$$\Pr(E_2 \mid F_1) \geq 1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n-1}.$$

Similarly,

$$\Pr(E_i \mid F_{i-1}) \geq 1 - \frac{k}{k(n-i+1)/2} = 1 - \frac{2}{n-i+1}.$$

To compute $\Pr(F_{n-2})$, we use

$$\begin{aligned}
\Pr(F_{n-2}) &= \Pr(E_{n-2} \cap F_{n-3}) = \Pr(E_{n-2} \mid F_{n-3}) \cdot \Pr(F_{n-3}) \\
&= \Pr(E_{n-2} \mid F_{n-3}) \cdot \Pr(E_{n-3} \mid F_{n-4}) \cdots \Pr(E_2 \mid F_1) \cdot \Pr(F_1) \\
&\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2} \left(\frac{n-i-1}{n-i+1}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{4}{6}\right)\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

■

Since the algorithm has a one-sided error, we can reduce the error probability by repeating the algorithm. Assume that we run the randomized min-cut algorithm $n(n-1)\ln n$ times and output the minimum size cut-set found in all the iterations. The probability that the output is not a min-cut set is bounded by

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\ln n} \leq e^{-2\ln n} = \frac{1}{n^2}.$$

In the first inequality we have used the fact that $1 - x \leq e^{-x}$.