Lecture 14: Randomness in algorithm design

Plan/outline

So far, all the algorithms we saw were *deterministic*, i.e., given an input, they take precisely the same amount of time, and produce precisely the same output. It turns out that for many problems, one can come up with much more effective algorithms when we allow *randomness* in the algorithm. Today's class will focus on giving some examples of this behavior.

Finding a frequent element in an array

Let us start with a toy example.

Problem. Let A be an (unsorted) array with n elements, and we are promised that for at least $n/3$ distinct indices i, $A[i] = 0$. The goal is to find some i such that $A[i] = 0$.

The trivial solution of iterating through the array to find such an index takes time $O(n)$ (as we may start at the beginning, and the 0 terms could all be located after the first $n/2$ positions.

Note, however, that if we select a *random* index i, the probability of having $A[i] = 0$ is at least $1/3$. Now, consider the following algorithm:

```
Procedure randomFind(A, r):
  pick 'r' random indices i_1, i_2, ..., i_r (with replacement)
  if A[i_t] = 0 for some t, output i_t
```

This process clearly only takes time $O(r)$ (assuming that generating a random index takes $O(1)$ time -- an assumption that we make for now). What can we say about the correctness of the procedure? On the one hand, it can fail, as none of the chosen indices satisfy $A[i_t] = 0$. But the probability of this happening is $(2/3)^r$ (because for each index, we have a success probability $\geq 1/3$, which means a failure probability $\leq 2/3$, and the indices are all independent). This decays exponentially with r, so for say $r = 100$, the failure probabilty is smaller than the probability of lightning striking any given point on earth at a given time. So for all practical purposes, this algorithm works much better than a linear scan -- it only takes $O(1)$ amount of time!

Remarks. The algorithm assumes that the array $A[]$ is already in memory, and we have random access. Assuming this, the algorithm does not even read the entire input! (In this sense, the algorithm is similar to sampling.) Also, by changing r in the algorithm, we also obtain a tradeoff between the success probability and the running time.

Checking identities

The problem is clearly a toy setting, but there are real problems that come remarkably close!

Problem: identity testing. Suppose we are given two polynomials $p(x)$ and $q(x)$ of degree d. Is $p(x) \equiv q(x)$? (I.e., is $p(x) = q(x)$ for all x?)

As an example, consider $p(x) = (x-7)(x-3)(x-1)(x+2)(2x+5)$ and $q(x) = 2x^5 - 13x^4 - 21x^3 + 127x^2 + 121x - 210$. The nice feature is that evaluating $p(x)$ and evaluating $q(x)$ can both be done relatively quickly, given an x. But expanding out $p(x)$ and matching with $q(x)$ can be cumbersome.

Generalizations of this problem to the multivariate case (say we have variables $x_1, \dots, x_r$) have important applications: consider two *circuits* (with input variables as leaves, and '+' and '*' operators in the internal nodes). Suppose the goal is to test if the circuits compute the same polynomial function of the inputs. Similarly, one might have different ``evaluation trees'', and we might be interested to know if they represent the same expression.

In all of these applications, we have a common problem: we are given polynomials p, q of degree d in r variables, such that *given some numeric values for the variables* $x_1, \dots, x_r$, $p(x_1, x_2, \dots, x_r)$ and $q(x_1, x_2, \dots, x_r)$ can be found in time poly(d, r). But expanding out and checking if $p \equiv q$ can take time exponential (in d or r).

Now, inspired by the topic of the day, we have the following algorithm: randomly choose values for $x_1, \dots, x_r$ in some large enough interval, and check if $p(x_1, x_2, \dots, x_r) = q(x_1, x_2, \dots, x_r)$. If the values are equal, we declare that $p \equiv q$, and if not, we declare that $p \not\equiv q$.

Now, if $p \equiv q$, the equality will always hold, and thus the algorithm outputs the right answer. The trouble is that even if $p \not\equiv q$, we might have chosen $x_1, \dots, x_r$ for which equality occurred for *those values*. Can we bound the probability of this happening?

Let us analyze in the simple case where we have a single variable x, and we have two degree d polynomials. Now, suppose we set x to be a random integer in the range $[1, 2d]$.

Claim. Let p, q be two degree d polynomials in one variable, and let y be a random integer in the range $[1, 2d]$. Then we have $\Pr[p(y) = q(y) \mid p \not\equiv q] \le 1/2$.

The claim says that the *test* above returns a false positive with probability $\le 1/2$.

Proof. Suppose $p \not\equiv q$. Then consider the polynomial $r(x) := p(x) - q(x)$. If p, q are of degree d, then r has degree $\le d$. Thus by basic algebra, we know that $r(x)$ has at most d roots. In particular, at most d of the $2d$ integers in the interval $[1, 2d]$ can be roots of $r(x) = 0$. Thus for a random y, $\Pr[r(y) = 0] \le 1/2$, which is equivalent to the claim.

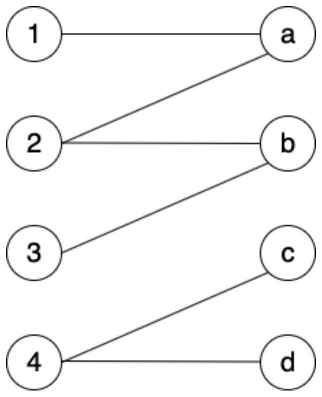
Generalization. A claim of this nature holds in much more generality, for polynomials of multiple variables, even on finite fields. This is commonly known as the *Schwartz-Zippel* lemma. The general problem, known as *polynomial identity testing*, is one of the fundamental problems in CS for which good probabilistic algorithms are known, but we do not know of efficient deterministic algorithms.

Perfect matching in a bipartite graph

We next see a non-trivial application of identity testing. Consider the following problem:

Problem. Given a bipartite graph (U, V, E) (vertex sets are U, V and edge set E) determine if it is possible to ``pair up'' **all** the vertices of U and V such that (a) every element of U is paired with precisely one element of V, and (b) i is paired with j only if $\{i, j\}$ is an edge (in E).

Clearly, for such a pairing to be possible, we need $|U| = |V|$ (otherwise some element of the larger set will be left unpaired). But this is not enough. E.g., consider the graph:



In this graph, the set {1,2,3} on the left is only connected to {a,b} on the right, so there is no way of pairing the vertices {1,2,3,4} with {a,b,c,d}. On the other hand, if we add the edge $\{1, d\}$ to the graph, we have a perfect matching -- (1, d), (2, a), (3, b), (4, c).

It turns out that the perfect matching problem can be solved in polynomial time using algorithms for the *max flow* problem, which we saw earlier. Let us now see a more "interesting" solution.

Consider an $n \times n$ matrix M (where $n = |V| = |U|$) whose rows correspond to vertices of U and columns to vertices of V. If $i \in U$ and $j \in V$ have an edge between them, then we place a variable called x_{ij} in the (i, j)th position of M. Otherwise, we place a 0 in the (i, j)th position of M. Thus, the matrix for the graph above would be:

$$M = \begin{pmatrix} x_{1a} & 0 & 0 & 0 \\ x_{2a} & x_{2b} & 0 & 0 \\ 0 & x_{3b} & 0 & 0 \\ 0 & 0 & x_{4c} & x_{4d} \end{pmatrix}.$$

Now the key idea is to consider the determinant of this matrix, denoted $\det(M)$. As the entries of M are variables (or zeros), $\det(M)$ is a polynomial in these variables (note that there are a total of $|E|$ variables). The main observation is now the following:

Claim. if the graph has a perfect matching, then $\det(M) \not\equiv 0$, and if G has no perfect matching, $\det(M) \equiv 0$.

Before we prove this claim, let us see why this implies a randomized algorithm for finding a perfect matching. Suppose we set the values of the variables to be random integers in some range (say $[1, 2n]$), then we can use the Schwartz-Zippel lemma to conclude that if $\det(M) \not\equiv 0$, then $\det(M)$ evaluated at the random integers is $\neq 0$ with probability $\geq 1/2$. On the other hand, if $\det(M) \equiv 0$, then the value will always be 0. Thus, consider the following algorithm:

```
procedure CheckPerfect(M, r):
  for j = 1, ..., r:
    set the variables in M to random values in the interval [1,2n]
    evaluate det(M) with these choice of variables
    if (det(M) != 0), return "graph has a perfect matching"
  end for
  // if test above fails for all j ...
  return "graph has no perfect matching
```

We can see that the algorithm outputs the correct answer with probability $\geq 1 - \frac{1}{2^r}$, and so the error probability drops exponentially as r grows. The running time is precisely r times the time taken to compute the determinant of an $n \times n$ matrix. It turns out that this is an extremely well-studied problem in numeric linear algebra -- the best known algorithm takes time $n^{2.38...}$ (same as the time taken for matrix multiplication). Thus the procedure is actually quite fast! (faster than most methods for the max flow problem).

With this motivation, let us prove the claim above.

Proof of claim. The claim turns out to be a consequence of a well-known formula for the determinant of a matrix: let A be an $n \times n$ matrix with entries a_{ij}. Then $\det(A) = \sum_{\text{permutations } \sigma \text{ of } \{1,2,...,n\}} (-1)^{\text{sign}(\sigma)} a_{1\sigma(1)} a_{2\sigma(2)} \dots a_{n\sigma(n)}$.

The sign of a permutation is defined as the number of ``transpositions'' -- the interested reader should refer to a good algebra textbook -- this definition of the determinant turns out to have some cool properties!

Once we understand this definition, the second part of the claim is immediate: if the graph has no perfect matching, then for any permutation σ, one of the $a_{i\sigma(i)}$ terms when we are evaluating $\det(M)$ would be zero, or a non-edge (otherwise the permutation defines a valid perfect matching!). Thus each of the terms is zero, implying that $\det(M) \equiv 0$.

Now if the graph does have a perfect matching (for simplicity, suppose it is $(1, A), (2, B), \dots$), then the term $x_{1A} x_{2B} \dots$ is a monomial that doesn't canceled by any other terms, and so this means that $\det(M) \not\equiv 0$. This proves the claim!

The argument is very elegant -- it uses the "permutation expansion" of the determinant (which has $n!$ terms) in order to prove the claim, and for actually evaluating the determinant when (random) numeric values are plugged in, the algorithm uses methods from numeric linear algebra to do it in $n^{2.38..}$ time.

Primality testing

Another early application of randomized algorithms is the "Miller-Rabin test" for primality. Proposed in the 1970s, the algorithm gives a randomized procedure that can test if an n-digit number is prime in time poly(n). Note that the standard "check divisors until sqrt of the number" procedure from middle-school takes time $2^{n/2}$ (because an n-digit number in binary is roughly of the order 2^n and its square root is $2^{n/2}$).

The Miller-Rabin test uses some interesting yet basic number theory. If you are interested in the details, see the wikipedia page, or the following notes which I thought were well-written:

Link to Sinclair's notes

Recap and comments

We saw some interesting examples of algorithms where (a) randomness was used, so two runs of the algorithm on the same input need not yield the same answer, (b) the algorithm need not always output the right answer, (c) as we increase the running time, the probability of correctness grows (in all the examples, the failure probability dropped exponentially).

More such trade-offs will appear in the next few lectures. We will also see another class of algorithms (called "Las Vegas algorithms"), where the algorithm is always *guaranteed to produce the right answer*, but the running time can depend on the random choices (i.e., sometimes the algorithm can be quite slow).