Advanced Algorithms Course. Lecture Notes. Part 11

Randomization Continued

Remark on the Complexity of Random Choices

Randomized algorithms make some random choices. That is, some random number is generated, and the next step of the algorithm depends on its value. In the randomized algorithm examples so far we did not bother about the time complexity of random number generation itself, because the time for that was included in the overall time bound. But in some other algorithms the random choices may abound, and they may dominate the overall time bound. Consequently, the time bound may depend on the model assumptions for random number generation.

A reasonable model is to count the number of **random bits** that are used. Random bits are independently set to 0 or 1 with probablity 1/2. For instance, selecting a random element from a set of n elements costs $\lceil \log_2 n \rceil$ random bits, generated in $O(\log n)$ time.

Chernoff Bounds

This is a very useful general tool to bound the probabilities that certain random variables deviate much from their expected values. Here we will derive one version of this bound and then apply it to a simple load balancing problem.

Let X be sum of n independent 0-1 valued random variables X_i taking value 1 with probability p_i. Clearly $E[X] = \sum_i p_i$. For $\mu := E[X]$ and $\delta > 0$ we ask how likely it is that $X > (1 + \delta)\mu$, in other words, that X exceeds the expected value by more than 100δ percent.

Since function exp is monotone, this inequality is equivalent to $\exp(tX) > \exp(t(1+\delta)\mu)$ for any $t > 0$. Exponentiation and this free extra parameter t

seem to make things more complicated, but we will see very soon why they are useful.

For any random variable Y and any number $\gamma > 0$ we have that $E[Y] \geq \gamma Pr(Y > \gamma)$. This is known as Markov's inequality and follows directly from the definition of $E[Y]$. For $Y := \exp(tX)$ and $\gamma = \exp(t(1+\delta)\mu)$ this yields $Pr(X > (1+\delta)\mu) \leq \exp(-t(1+\delta)\mu)E[\exp(tX)]$.

Due to independence of the terms X_i we have

$$E[\exp(tX)] = E[\exp(\sum_i tX_i)] = E[\prod_i \exp(tX_i)] = \prod_i E[\exp(tX_i)]$$

$$= \prod_i (p_i e^t + 1 - p_i) = \prod_i (1 + p_i(e^t - 1)) \leq \prod_i \exp(p_i(e^t - 1))$$

$$= \exp\left((e^t - 1)\sum_i p_i\right) \leq \exp((e^t - 1)\mu).$$

This gives us the bound $\exp(-t(1+\delta)\mu)\exp((e^t-1)\mu)$. We can arbitrarily choose t. With $t := \ln(1+\delta)$ our bound reads as $\left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$.

The base depending on δ looks a bit complicated, however: Using $e^\delta \approx 1 + \delta$ one can see that the base is smaller than 1. For any fixed deviation δ the base is constant, and the bound decreases exponentially in μ. The more independent summands X_i we have in X, the smaller is the probability of large deviations. A direct application of the simple Markov inequality would be much weaker (therefore the detour via the exponential function).

Load Balancing

In order to show at least one application, consider the following simple load balancing problem: m jobs shall be assigned to n processors, in such a way that no processor gets a high load. In contrast to the Load Balancing problem we studied earlier, no central "authority" assigns jobs to processors, but every job chooses a processor by itself. We want to install a simple rule yet obtain a well balanced allocation. (An application is distributed processing of independent tasks in networks.) To make the rule as light-weight as possible, let us choose for every job a processor randomly and independently. The jobs need not even "talk" to each other and negotiate places. How good is this policy?

We analyze only the case $m = n$. What would you guess: How many jobs end up on the same processor? To achieve clarity, consider the random variable X_i defined as the number of jobs assigned to processor i. Clearly $E[X_i] = 1$. The quantity we are interested in is $Pr(X_i > c)$, for a given bound c. Since X_i is a sum of independent 0-1 valued random variables (every job chooses processor i or not), we can apply the Chernoff bound. With $\delta = c - 1$ and $\mu = 1$ we get immediately the bound $e^{c-1}/c^c < (e/c)^c$.

But this is only the probability bound for one processor. To bound the probability that $X_i > c$ holds for *some* of the n processors, we can apply the union bound and multiply the above probability with n. Now we ask: For which c will $n(e/c)^c$ be "small"?

At least, we must choose c large enough to make $c^c > n$. As an auxiliary calculation consider the equation $x^x = n$. For such x we can say:
(1) $x \log x = \log n$ and
(2) $\log x + \log\log x = \log\log n$.
Here we have just taken the logarithm twice. Equation (2) easily implies

$$\log x < \log\log n < 2\log x.$$

Division by (1) yields

$$1/x < \log\log n/\log n < 2/x.$$

In other words, $x^x = n$ holds for some $x = \Theta(\log n/\log\log n)$.

Thus, if we choose $c := ex$, our Chernoff bound for every single processor simplifies to $1/x^{ex} < 1/(x^x)^2 = 1/n^2$. This finally shows: With probability $1 - 1/n$, each processor gets $O(\log n/\log\log n)$ jobs. This answers our question: Under random assignments, the maximum load can be logarithmic, but it is unlikely to be worse.

For $m = \Theta(n \log n)$ or more jobs, the random load balancing becomes really good. Then the load is larger than twice the expected value $\Theta(\log n)$ only with probability below $1/n^2$. Calculations are similar as above.

Verifying a Matrix Product

Randomized algorithms are surprisingly simple and powerful for many problems, however they come with only probabilistic "guarantees". A Las Vegas algorithm may be fast on expectation, but in a particular case we may have to wait longer for a result, which can be criticial in real-time application.

A Monte Carlo algorithm can err with some small but positive probablity. Maybe this means only a slightly worse result, but maybe it has desastrous consequences if the unlikely case happens. Then we have to judge whether the risk is acceptable. This does not depend so much on the mathematical problem solved, but on the real-world context where the algorithm is applied.

Amzingly, randomization can also lead to more safety: Even the result of a complex deterministic calculation can be false due to hardwre failure, a corrupted file or transmission errors. If accuracy is very important, it would be good to efficiently verify the result afterwards.

A famous example is Freivald's verifier for matrix multiplication. Many technical calculations use linear algebra, and matrix multiplication is a basic operation there. Let A and B be two $n \times n$ matrices. Suppose that we got their product C and want to check its correctness. The naive idea is to recalculate AB and compare with C. But matrix multiplication costs $O(n^3)$ time. There exist subcubic algorithms, but they are barely practical. In any case, significantly more than quadratic time is needed.

The idea for fast verification is to check whether $ABx = Cx$ for some vector x. Note that this requires only $O(n^2)$ time, since only matrix-vector multiplications are involved: We first compute the vector Bx and then $A(Bx)$. If really $AB = C$ then, obviously, we get $ABx = Cx$. The converse is not true: We may "incidentally" observe $ABx = Cx$ although $AB \neq C$. But how likely is this event? Specifically, let x be a vector whose entries are 0 or 1 (the real numbers, not Boolean values), independently and with probability 1/2. Assume $AB \neq C$, hence the matrix $D := AB - C$ has some nonzero entry, without loss of generality in the first row and column. Let d^T denote the first row of D. Let d' and x' be the vector d and x, without the first entries $d_1 \neq 0$ and x_1, respectively. Then the first entry of Dx equals $d^T x = d_1 x_1 + d'^T x'$. For any fixed choice of x', the second term is constant. Now remember that the x_i are independent. Thus x_1 is still 0 or 1 with conditional probabilty 1/2. Moreover, since $d_1 \neq 0$, at least one of these cases yields $d^T x \neq 0$. We conclude that a false C passes the test with probability at most 1/2. Finally we can repeat this $O(n^2)$ time test with t independent vectors x to reduce the error probability to $1/2^t$.