15-251: Great Theoretical Ideas In Computer Science

Recitation 8: Randomization (and Games and Approximations) Solutions

Lecture Review

- A randomized algorithm is an algorithm that has access to random bits, i.e. it can flip a coin. In this class we will allow randomized algorithms to call $\text{RandInt}(n)$ and $\text{Bernoulli}(p)$.

- An algorithm $A$ is a $T(n)$-time Las Vegas algorithm if
  - $A$ always outputs the right answer, and
  - for every input $x \in \Sigma^*$, $\mathbb{E}[\text{number of steps } A(x) \text{ takes}] \leq T(|x|)$.

- An algorithm $A$ is a $T(n)$-time Monte Carlo algorithm with error probability $\varepsilon$ if
  - for every input $x \in \Sigma^*$, $A(x)$ gives the wrong answer with probability at most $\varepsilon$, and
  - for every input $x \in \Sigma^*$, $A(x)$ has a worst-case running-time of at most $T(|x|)$.

- Boosting lets one improve the success probability of Monte Carlo algorithms via repeated trials.

A Hard Exam

(a) Suppose that the average score on the latest 15-150 exam was 10 points out of 100 and that 200 students took the exam. What’s an upper bound on the number of students who received a perfect score? Assume that the 150 TAs are kind enough to not assign negative scores to students.

It’s twenty students.

Let $k$ be the number of students who received a perfect score.

If $k > 20$ students received a perfect score, then the average score is at least \( \frac{100k}{200} = \frac{k}{2} > 10 \).

(Here we used the fact that exam scores are non-negative.) By contrapositive, we have $k \leq 20$.

(Moreover, this bound is tight. We can achieve $k = 20$ if twenty students receive a perfect score and everyone else receives a score of zero.)

(b) Markov’s inequality: Let $X$ be a non-negative random variable with non-zero expectation. For any $c > 0$,

$$\Pr[X \geq c \mathbb{E}[X]] \leq \frac{1}{c}.$$  

(No need to prove this — refer to the course notes to see the proof. But note that the proof is similar to the reasoning in part (a).)

What happens in Las Vegas doesn’t stay in Monte Carlo

The expected number of comparisons that the Quicksort algorithm makes is at most $2n \ln n$ (which you can cite without proof — you might see a proof of this fact if you take 15-210). Describe how to convert this Las Vegas algorithm into a Monte Carlo algorithm with the worst-case number of comparisons being $1000n \ln n$. Give an upper bound on the error probability of the Monte Carlo algorithm.
First attempt: Run the Las Vegas algorithm, but if it performs $1000n \ln n$ comparisons, we will stop the algorithm and declare failure. Let $X$ be a random variable for the number of comparisons performed a run of the Las Vegas algorithm. Our algorithm only fails if the Las Vegas algorithm tries to perform $\geq 1000n \ln n$ comparisons. By Markov's inequality, this occurs with probability

$$\Pr[X \geq 1000n \ln n] \leq \frac{1}{500}.$$ 

This is not really a great bound, so let’s try something else.

Second attempt: Now we will use boosting to achieve a much better bound for the error probability. Run the Las Vegas algorithm, but if it performs $4n \ln n$ comparisons, stop the algorithm and declare failure. Again, by Markov’s inequality, this occurs with probability at most $1/2$. We will do this independently 250 times (so we have at $1000n \ln n$ comparisons in total). If any repetition succeeds, we can give its output and be correct. This way only fail if all 250 repetitions fail, which occurs with probability $(1/2)^{250} < 1/10^{75}$ by independence.

Randomization Meets Approximation

3SAT is difficult to solve exactly, but it turns out it’s not hard to find a decent approximation.

Consider the MAX-3SAT problem where, given a CNF formula in which every clause has exactly 3 literals (with distinct variables), we want to find a truth assignment to the variables in the formula so that we maximize the number of clauses that evaluate to True.

Describe a polynomial-time randomized algorithm with the property that, given a 3CNF formula with $m$ clauses, it outputs a truth assignment to the variables such that the expected number of clauses that evaluate to True is $\frac{7}{8}m$ (i.e., in expectation, the algorithm is a $\frac{7}{8}$-approximation algorithm).

The solution here is simply to pick the assignment to the variables randomly: for each variable, flip a coin. If it is heads, assign it the value True. If it is tails, assign it False. Suppose we have $m$ clauses. Now let $X$ be the number of clauses satisfied. We are interested in computing $E[X]$. Define the indicator random variable $X_i$ to be 1 if the $i$’th clause is satisfied, and 0 otherwise. Then

$$X = \sum_{i=1}^{m} X_i$$

and so

$$E[X] = E[\sum_{i=1}^{m} X_i] = \sum_{i=1}^{m} E[X_i]$$

where in the last equality, we used linearity of expectation. Note that for all $i$,

$$E[X_i] = \Pr[X_i = 1] = \frac{7}{8}$$

the only way a clause is not satisfied is when all the literals are False, which happens with probability $(1/2)^3 = 1/8$. So $E[X] = \frac{7}{8}m$.

Here it is important to note that that the $X_i$’s need not be independent! In fact, they can be perfectly correlated: consider a case where all the $m$ clauses are identical to each other. Then if you know that one clause is satisfied, you automatically know that all of the clauses must be satisfied. Yet, this does not change the expected number of satisfied clauses at all. And as you can see in the calculation above, linearity of expectation allows us to treat each clause independently, without worrying about potential dependencies among clauses. We know that $E[X_i] = \Pr[X_i = 1] = 7/8$ no matter what is going on with other clauses.
(Bonus) Splitting Differences

Consider a variant of Nim, in which the game begins with a single pile of \( n \) stones, and a player performs one of the following actions on their turn:

- Take any number of stones from a pile.
- Split any pile into two non-empty piles.

As usual, the player who makes the last move wins. Compute the Nimbers for this game.

We have

\[
\begin{align*}
N(4k + 1) &= 4k + 1 \\
N(4k + 2) &= 4k + 2 \\
N(4k + 3) &= 4k + 4 \\
N(4k + 4) &= 4k + 3
\end{align*}
\]

This is a straightforward proof by induction.

(Bonus) Cut and Dried

We define the Max-Cut problem as follows:

Let \( G = (V, E) \) be a graph. Given a coloring of the vertices with 2 colors, we say that an edge \( e = \{u, v\} \) is cut if \( u \) and \( v \) are colored differently. In the Max-Cut problem, the input is a graph \( G \), and the output is a coloring of the vertices with 2 colors that maximizes the number of cut edges.

Consider the following approximation algorithm for the Max-Cut problem:

\[
\text{function } \text{MaxCutApprox}(G = (V, E)) \\
\text{color all vertices red} \\
\text{\hspace{1cm}} \triangleright \text{allowable colors are red and something else (say, blue)} \\
\text{while there exists } v \in V \text{ s.t. changing } v \text{'s color increases the number of cut edges do} \\
\text{\hspace{3cm}} \text{change the color of } v \\
\text{end while} \\
\text{return coloring} \\
\text{end function}
\]

(a) Show that this algorithm is poly-time.

(b) Prove that this algorithm is a \( \frac{1}{2} \)-approximation for Max-Cut.

(c) Show that this algorithm is not a \( (\frac{1}{2} + \varepsilon) \)-approximation algorithm for Max-Cut for any \( \varepsilon > 0 \).
See Chapter 13 for a solution to (a) and (b).

Towards (c), let \( k \) be arbitrary. Informally, draw \( k \) copies of \( C_6 \) arranged in a line, and then connect all middle vertices in a path (an example when \( k = 3 \) has been drawn below). A poor algorithm trace colors all the vertices on the path red and the rest of the vertices blue, achieving \( 4k \) cut edges; it is easy to verify that it is possible for the algorithm to do this, and once it reaches this point it will terminate. Observe that there are \( 8k - 1 \) edges in total, and since the graph is bipartite the optimal solution will cut all the edges. So the approximation ratio for this algorithm cannot exceed \( \frac{4k}{8k - 1} \); taking \( k \) sufficiently large shows that this is not a \( \left( \frac{1}{2} + \varepsilon \right) \)-approximation for any constant \( \varepsilon > 0 \).

![Diagram of \( k \) copies of \( C_6 \) arranged in a line, with middle vertices connected in a path.](image)

(Extra) Passive-Aggressive Passengers

Consider a plane with \( n \) seats \( s_1, s_2, \ldots, s_n \). There are \( n \) passengers, \( p_1, p_2, \ldots, p_n \) and they are randomly assigned unique seat numbers. The passengers enter the plane one by one in the order \( p_1, p_2, \ldots, p_n \). The first passenger \( p_1 \) does not look at their assigned seat and instead picks a uniformly random seat to sit in. All the other passengers, \( p_2, p_3, \ldots, p_n \), use the following strategy. If the seat assigned to them is available, they sit in that seat. Otherwise they pick a seat uniformly at random among the available seats, and they sit there. What is the probability that the last passenger, \( p_n \), will end up sitting in their assigned seat?

For ease of presentation, note that the random assignment of seats is a red herring — we can without loss of generality rearrange seat numbers so that \( p_i \) is assigned to \( s_i \).

We take a different perspective on the problem as follows: The passengers \( p_2, p_3, \ldots, p_{n-1} \) use the following strategy. Passenger \( p_i \) goes to their assigned seat \( s_i \). If it is occupied, they kick out the intruder (who will always be \( p_1 \)). No matter what, \( p_i \) sits in \( s_i \). Then the intruder picks a uniformly random available seat to sit in. This is the same as the original problem, but the identities are changed around so that \( p_1 \) is the only passenger going around picking uniformly random seats. This perspective makes the situation a little easier to think about. Now we want to find the probability that \( s_n \) is unoccupied when \( p_n \) comes to take a seat.

Notice that it only matters whether \( p_1 \) eventually sits in \( s_1 \) or in \( s_n \). If \( p_1 \) sits elsewhere, they will be kicked out and have to repick their seat until eventually they choose \( s_1 \) or \( s_n \). Then \( p_1 \) won’t move again until \( p_n \) comes, at which point \( s_n \) is unoccupied iff \( p_1 \) sits in \( s_1 \). Every time \( p_1 \) picks a seat, they have an equal chance of going to \( s_1 \) and of going to \( s_n \), so the answer is \( \frac{1}{2} \).