1.8 Lecture 8 (19/Oct) Part I: Achieving expectation in MAX-3SAT.

Logistics: Jenish's OH will be in Annenberg 107 or 121 according to the day; see class webpage and Google calendar.

1.8.1 Another appetizer

Consider unbiased random walk on the n-cycle. Index the vertices clockwise $0, \ldots, n-1$, and start the walk at 0. What is the probability distribution on the *last* vertex to be reached?

1.8.2 MAX-3SAT

Let's start looking at some computational problems. A 3CNF formula on variables $x_1, \ldots, x_n$ is the conjunction of clauses, each of which is a disjunction of at most three literals. (A literal is an x_i or x_i^c, where x_i^c is the negation of x_i.)

You will recall that it is NP-complete to decide whether a 3CNF formula is satisfiable, that is, whether there is an assignment to the x_i's s.t. all clauses are satisfied. Let's take a little different focus: think about the *maximization* problem of satisfying as many clauses as possible. Of course this is NP-hard, since it includes satisfiability as a special case. But, being an optimization problem, we can still ask how well we can do.

Theorem 22 *For any 3CNF formula there is an assignment satisfying $\geq 7/8$ of the clauses. Moreover such an assignment can be found in randomized time $O(m^2)$, where m is the number of clauses (and we suppose that every variable occurs in some clause).*

Proof: The existence assertion is due to linearity of expectation, while the algorithm might be attributed to the English educator Hickson [51]: *'Tis a lesson you should heed: / Try, try, try again. / If at first you don't succeed, / Try, try, try again.* Now that we've been suitably educated, let's ask, how long does this process take? In a single trial we check one assignment, which takes time $O(m)$. How many trials do we need to succeed?

Let the rv M be the number of satisfied clauses of a random assignment. $m - M$ is a nonnegative rv with expectation $m/8$, and Markov's inequality tells us that $\Pr(M \leq (7/8 - \varepsilon)m) = \Pr(m - M \geq (1 + 8\varepsilon)m/8) \leq 1/(1 + 8\varepsilon)$.

This says we have a good chance of getting close to the desired number of satisfied clauses; however, we asked to achieve $7/8$, not $7/8 - \varepsilon$. We can get this by noting that M is integer-valued, so for $\varepsilon < 1/m$, an assignment satisfying $7/8 - \varepsilon$ of the clauses, satisfies $7/8$ of them.

With the choice $\varepsilon = \frac{1}{2m}$, then, the probability that a trial succeeds is at least

$$1 - \frac{1}{1 + 8\varepsilon} = \frac{8\varepsilon}{1 + 8\varepsilon} = \frac{4}{m + 4} \in \Omega(1/m)$$

Trials succeed or fail independently so the expected number of trials to success is the expectation of a geometric random variable with parameter $\Omega(1/m)$, which is $O(m)$. □

1.8.3 Derandomization by the method of conditional expectations

How can we improve on this simple-minded method? We do not have a way forward on increasing the fraction of satisfied clauses, because of:

Theorem 23 (Håstad [49]) *For all $\varepsilon > 0$ it is NP-hard to approximate Max-3SAT within factor $7/8 + \varepsilon$.*

But we might hope to reduce the runtime, and also perhaps the dependence on random bits. As it turns out we can accomplish both of these objectives.

Theorem 24 *There is an $O(m)$-time deterministic algorithm to find an assignment satisfying $7/8$ of the clauses of any 3CNF formula on m clauses.*

Proof: This algorithm illustrates the *method of conditional expectations*. The point is that we can derandomize the randomized algorithm by not picking all the variables at once. Instead, we consider the alternative choices to just one of the variables, and choose the branch on which the conditional expected number of satisfying clauses is greater.

This very general method works in situations in which one can actually quickly calculate (or at least approximate) said conditional expectations.

We use the tower property of conditional expectations, (1.22): letting $Y =$ the number of satisfied clauses for a uniformly random setting of the rvs,

$$E(Y) = E(E(Y|x_1))$$

or explicitly

$$E(Y) = \frac{1}{2}E(Y|x_1 = 0) + \frac{1}{2}E(Y|x_1 = 1)$$

and the strategy is to pursue the value of x_1 which does better.

In the present example computing the conditional expectations is easy. The probability that a clause of size i is satisfied is $1 - 2^{-i}$. If a formula has m_i clauses of size i, the expected number of satisfied clauses is $\sum m_i(1 - 2^{-i})$. Now, partition the clauses of size i into m_i^1 that contain the literal x_1, m_i^0 that contain the literal x_1^c, and those that contain neither.

The expected number of satisfied clauses conditional on setting $x_1 = 1$ is

$$\sum m_i^1 + \sum m_i^0(1 - 2^{-i+1}) + \sum(m_i - m_i^1 - m_i^0)(1 - 2^{-i}). \tag{1.24}$$

Similarly the expected number of satisfied clauses conditional on setting $x_1 = 0$ is

$$\sum m_i^1(1 - 2^{-i+1}) + \sum m_i^0 + \sum(m_i - m_i^1 - m_i^0)(1 - 2^{-i}). \tag{1.25}$$

A simple way to do this: we can compute each of these quantities in time $O(m)$. (Actually, since these quantities average to the current expectation, which we already know, we only have to calculate one of them.) This simple process runs in time $O(m^2)$. However, we can actually do the

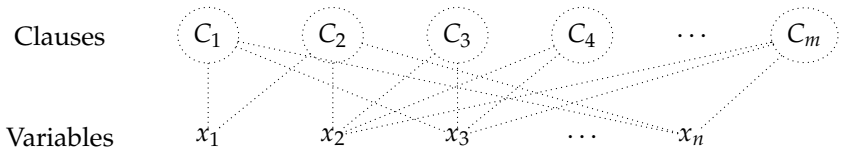

Figure 1.3: m clauses of size ≤ 3, n variables

process in time $O(m)$. We don't even really need to calculate the quantities (1.24),(1.25). We start with variable x_1 and scan all the clauses it participates in (see Fig. 1.3). For each clause C_i (which say has currently $|C_i|$ literals), the effect of setting $x_1 = 1$ change the contribution of the clause

to the expectation from $1 - 2^{-|C_i|}$ to either 1 (if the variable satisfies the clause) or to $1 - 2^{-|C_i-1|}$ (otherwise); i.e., the expectation either increases or decreases by $2^{-|C_i|}$, while the effect of setting $x_1 = 0$ is exactly the negative of this. We add these contributions of $\pm 2^{-|C_i|}$, conditional on $x_1 = 1$, as we scan the clauses containing x_1; if it is nonnegative we fix $x_1 = 1$, otherwise we fix $x_1 = 0$. Having done that, we edit the relevant clauses to eliminate x_1 from them. Then we continue with x_2, etc. The work spent per variable is proportional to its degree in this bipartite graph (the number of clauses containing it), and the sum of these degrees is $\leq 3m$. So the total time spent is $O(m)$. □

Chapter 2

Algebraic Fingerprinting

There are several key ways in which randomness is used in algorithms. One is to "push apart" things that are different even if they are similar. We'll study a few examples of this phenomenon.

2.1 Lecture 8 (19/Oct) Part II: Fingerprinting with Linear Algebra

2.1.1 Polytime Complexity Classes Allowing Randomization

Some definitions of one-sided and two-sided error in randomized computation are useful.

Definition 27 *BPP, RP, coRP, ZPP: These are the four main classes of randomized polynomial-time computation. All are decision classes. A language L is:*

- *In BPP if the algorithm errs with probability* $\leq 1/3$.
- *In RP if for* $x \in L$ *the algorithm errs with probability* $\leq 1/3$, *and for* $x \notin L$ *the algorithm errs with probability* 0.

(note, RP is like NP in that it provides short proofs of membership), while the subsidiary definitions are:

- $L \in$ *coRP if* $L^c \in RP$, *that is to say, if for* $x \notin L$ *the algorithm errs with probability* $\leq 1/3$, *and for* $x \in L$ *the algorithm errs with probability* 0.
- $ZPP = RP \cap coRP$.

It is a routine exercise that none of these constants matter and can be replaced by any $1/poly$, *although completing that exercise relies on the Chernoff bound which we'll see in a later lecture.*

Exercise: Show that the following are two equivalent characterizations of ZPP: (a) there is a poly-time randomized algorithm that with probability $\geq 1/3$ outputs the correct answer, and with the remaining probability halts and outputs "don't know"; (b) there is an expected-poly-time algorithm that always outputs the correct answer.

We have the following obvious inclusions:

$$\mathrm{P} \subseteq \mathrm{ZPP} \subseteq \mathrm{RP}, \mathrm{coRP} \subseteq \mathrm{BPP}$$

What is the difference between ZPP and BPP? In BPP, we can never get a definitive answer, no matter how many independent runs of the algorithm execute. In ZPP, we can, and the expected time until we get a definitive answer is polynomial; but we cannot be sure of getting the definitive answer in any fixed time bound.

Here are the possible outcomes for any single run of each of the basic types of algorithm:

	$x \in L$	$x \notin L$
RP	$\in, \notin$	$\notin$
coRP	$\in$	$\in, \notin$
BPP	$\in, \notin$	$\in, \notin$

If $L \in ZPP$, then we can be running simultaneously an RP algorithm A and a coRP algorithm B for L. Ideally, this will soon give us a definitive answer: if both algorithms say "$x \in L$", then A cannot have been wrong, and we are sure that $x \in L$; if both algorithms say "$x \notin L$", then B cannot have been wrong, and we are sure that $x \notin L$. The expected number of iterations until one of these events happens (whichever is viable) is constant. But, in any particular iteration, we *can* (whether $x \in L$ or $x \notin L$) get the indefinite outcome that A says "$x \notin L$" and B says "$x \in L$". This might continue for arbitrarily many rounds, which is why we can't make any guarantee about what we'll be able to prove in bounded time.

An algorithm with "BPP"-style two-sided error is often referred to as "Monte Carlo", while a "ZPP"-style error is often referred to as "Las Vegas".

2.1.2 Verifying Matrix Multiplication

It is a familiar theme that *verifying* a fact may be easier than *computing* it. Most famously, it is widely conjectured that P≠NP. Now we shall see a more down-to-earth example of this phenomenon.

In what follows, all matrices are $n \times n$. In order to eliminate some technical issues (mainly numerical precision, also the design of a substitute for uniform sampling), we suppose that the entries of the matrices lie in $\mathbb{Z}/p$, p prime; and that scalar arithmetic can be performed in unit time.

(The same method will work for any finite field and a similar method will work if the entries are integers less than $\text{poly}(n)$ in absolute value, so that we can again reasonably sweep the runtime for scalar arithmetic under the rug.)

Here are two closely related questions:

1. Given matrices A, B, *compute* $A \cdot B$.
2. Given matrices A, B, C, *verify* whether $C = A \cdot B$.

The best known algorithm, as of 2014, for the first of these problems runs in time $O(n^{2.3728639})$ [42]. Resolving the correct lim inf exponent (usually called ω) is a major question in computer science.

Clearly the second problem is no harder, and a lower bound of $\Omega(n^2)$ even for that is obvious since one must read the whole input.

Randomness is not known to help with problem (1), but the situation for problem (2) is quite different.

Theorem 28 (Freivalds [39]) *There is a coRP-style algorithm for the language "$C = A \cdot B$", running in time $O(n^2)$.*

I wrote "coRP-style" rather than coRP because the issue at stake is not the polynomiality of the runtime (since $n^{\omega+o(1)}$ is an upper bound and the gain from randomization is that we're achieving n^2), but only the error model.

Proof: Note that the obvious procedure for matrix-vector multiplication runs in time $O(n^2)$.

The verification algorithm is simple. Select uniformly a vector $x \in (\mathbb{Z}/p)^n$. Check whether $ABx = Cx$ without ever multiplying AB: applying associativity, $(AB)x = A(Bx)$, this can be done in just three matrix-vector multiplications. Output "Yes" if the the equality holds; output "No" if it fails. Clearly if $AB = C$, the output will be correct. In order to get a coRP-style result, it remains to show that

$$\Pr(ABx = Cx | AB \neq C) \leq 1/2.$$

The event $ABx = Cx$ is equivalently stated as the event that x lies in the right kernel of $AB - C$. Given that $AB \neq C$, that kernel is a *strict* subspace of $(\mathbb{Z}/p)^n$ and therefore of at most half the cardinality of the larger space. Since we select x uniformly, the probability that it is in the kernel is at most $1/2$. □