Lecture 19

Prof. Nick Harvey *University of British Columbia*

Today's lecture introduces a completely new topic: the Lovász Local Lemma (LLL). This is an important method for analyzing events that are not independent, but have some restricted sort of dependencies. It is not as widely applicable as many of the other the techniques we have seen so far, but from time to time one encounters scenarios in which the LLL is the only technique that works.

1 The Lovász Local Lemma

Suppose $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are a collection of "bad" events. We would like to show that there is positive probability that none of them occur. If the events are mutually independent then this is simple:

$$\Pr[\, \wedge_{i=1}^n \overline{\mathcal{E}_i} \,] \;=\; \prod_{i=1}^n \Pr[\overline{\mathcal{E}_i}] \;>\; 0$$

(assuming that $\Pr[\mathcal{E}_i] < 1$ for every i). The LLL is a method for proving that $\Pr[\wedge_{i=1}^n \overline{\mathcal{E}_i}] > 0$ when the $\mathcal{E}_i$'s are not mutually independent, but they can have some sort of limited dependencies.

Formally, we say that an event $\mathcal{E}_j$ does not depend on the events $\{\mathcal{E}_i : i \in I\}$ if

$$\Pr[\mathcal{E}_j] \;=\; \Pr[\, \mathcal{E}_j \mid \wedge_{i \in I'} \mathcal{E}_i \,] \qquad \forall I' \subseteq I.$$

So, regardless of whether some of the events in I occur, the probability of $\mathcal{E}_j$ occurring is unaffected.

Theorem 1 (The "Symmetric" LLL) *Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be events with $\Pr[\mathcal{E}_i] \le p$ for all i. Suppose that every event $\mathcal{E}_i$ does not depend on at least $n - d$ other events. If $pd \le 1/e$ then $\Pr[\wedge_{i=1}^n \overline{\mathcal{E}_i}] > 0$.*

We will not prove this theorem. Instead, we will illustrate the LLL by considering a concrete application of it in showing satisfiability of k-CNF Boolean formulas. Recall that a k-CNF formula is a Boolean formula, involving any finite number of variables, where the formula is a *conjunction* ("and") of any number of clauses, each of which is a *disjunction* ("or") of exactly k distinct *literals* (a variable or its negation).

For example, here is a 3-CNF formula with three variables and eight clauses.

$$\begin{aligned} \phi(a,b,c) \;=\; & (a \vee b \vee c) \wedge (a \vee b \vee \bar{c}) \wedge (a \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c}) \wedge \\ & (\bar{a} \vee b \vee c) \wedge (\bar{a} \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee \bar{c}) \end{aligned}$$

This formula is obviously unsatisfiable. One can easily generalize this construction to get an unsatisfiable k-CNF formula with k variables and 2^k clauses. Our next theorem says: the reason this formula is unsatisfiable is that we allowed each variable to appear in too many clauses.

Theorem 2 *There is a universal constant $d \ge 1$ such that the following is true. Let ϕ be a k-CNF formula where each variable appears in at most $T := 2^{k-d}/k$ clauses. Then ϕ is satisfiable. Moreover, there is a randomized, polynomial time algorithm to find a satisfying assignment.*

The theorem is stronger when d is small. The proof that we will present can be optimized to get $d = 3$. By applying the full-blown LLL one can achieve $d = \log_2(e) \approx 1.44$.

Let n be the number of variables and m be the number of clauses in ϕ. Each clause contains k variables, each of which can appear in only $T - 1$ other clauses. So each clause shares a variable with less than $R := kT = 2^{k-d}$ other clauses.

The algorithm proving the theorem is perhaps the most natural algorithm that one could imagine. However it took more than 30 years from the introduction of the LLL for this algorithm to be provably analyzed.

SOLVE(ϕ)

- Set each variable in ϕ to either 0 or 1 randomly and independently.
- While there is an unsatisfied clause C
- FIX(C)

FIX(C)

- Set each variable in C to either 0 or 1 randomly and independently.
- While there is an unsatisfied clause D sharing some variable with C (possibly $D = C$)
- FIX(D)

Claim 3 *Suppose every call to* FIX *terminates. Then* SOLVE *calls* FIX *at most m times, and terminates with a satisfying assignment.*

PROOF: For any call to FIX, we claim that every clause that was satisfied *before* the call is still satisfied *after* the call completes. This follows by induction, starting at the deepest level of recursion. So, for every call from SOLVE to FIX(C) the number of satisfied clauses increases by one, since C must now be satisfied when FIX(C) terminates. □

So it remains to show that, with high probability, every call to FIX terminates.

Theorem 4 *Let $s = m(\log m + c) + \log \frac{1}{\delta}$ where c is a sufficiently large constant. Then the probability that the algorithm makes more than s calls to* FIX *(including both the top-level and recursive calls) is at most δ.*

The proof proceeds by considering the interactions between two agents: the "CPU" and the "Debugger". The CPU runs the algorithm, periodically sending messages to the Debugger (we describe these messages in more detail below). However, if FIX gets called more than s times the CPU interrupts the execution and halts the algorithm.

The CPU needs n bits of randomness to generate the initial assignment in SOLVE, and needs k bits to regenerate variables in each call to FIX. Since the CPU will not execute FIX more than s times, it might as well generate all its random bits at the very start of the algorithm. So the first step performed by the CPU is to generate a random bitstring x of length $n + sk$ to provide all the randomness used in executing the algorithm.

The messages sent from the CPU to the Debugger are as follows.

- Every time the CPU runs FIX(C), he sends a message containing the identity of the clause C, and an extra bit indicating whether this is a top-level FIX (i.e., a call from SOLVE) or a recursive FIX.
- Every time FIX(C) finishes the CPU sends a message stating "recursive call finished".
- If FIX gets called s times, the CPU sends a message to the Debugger containing the current $\{0,1\}$ assignment of all n variables.

Because the Debugger is notified when every call to FIX starts or finishes, he always knows which clause is currently being processed by FIX. A crucial detail is to figure out how many bits of communication are required to send these messages.

- For a top-level FIX, $\log m + O(1)$ bits suffice because there are only m clauses in ϕ.
- For a recursive FIX, $\log R + O(1)$ bits suffice because the Debugger already knows what clause is currently being fixed, and that clause shares variables with only R other clauses, so only R possible clauses could be passed to the next call to FIX.
- When each call to FIX(C) finishes, the corresponding message takes $O(1)$ bits.
- When FIX gets called s times, the corresponding message takes $n + O(1)$ bits.

The main point of the proof is to show that, if FIX gets called s times, then these messages reveal the random string x to the Debugger.

Since each clause is a *disjunction* (an "or" of k literals), there is *exactly one* assignment to those variables that does not satisfy the clause. So, whenever the CPU tells the Debugger that he is calling FIX(C), the Debugger knows exactly what the current assignment to C is. So, starting from the assignment that the Debugger received in the final message, he can work backwards and figure out what the previous assignment was before calling FIX. Repeating this process, he can figure out how the variables were set in each call to FIX, and also what the initial assignment was. Thus the Debugger can reconstruct the random string x.

The total number of bits sent by the CPU are

- $m(\log m + O(1))$ bits for all the messages sent when SOLVE calls FIX.
- $s \cdot (\log R + O(1))$ for all the messages sent in the $\leq s$ recursive calls.
- $n + O(1)$ bits to send the final assignment.

So x has been compressed from $n + sk$ bits to

$$m(\log m + O(1)) \;+\; s(\log R + O(1)) \;+\; n + O(1) \quad \text{bits.}$$

This is an overall shrinking of

$$\begin{aligned}
& \Big(n + sk\Big) \;-\; \Big(m(\log m + O(1)) \;+\; s(\log R + O(1)) \;+\; n + O(1)\Big) \\
= \;& s(k - \log R - O(1)) \;-\; m(\log m + O(1)) \;-\; O(1) \\
= \;& s(d - O(1)) \;-\; m(\log m + O(1)) \qquad (\text{since } R = 2^{k-d}) \\
= \;& \big(m(\log m + c) + \log \tfrac{1}{\delta}\big)(d - O(1)) \;-\; m(\log m + O(1)) \qquad (\text{definition of } s) \\
\geq \;& \log \frac{1}{\delta}
\end{aligned}$$

bits, assuming that c and d are sufficiently big constants.

We have argued that, if FIX gets called s times, then x can be compressed by $\log \frac{1}{\delta}$ bits. The next claim argues that this happens with probability at most δ.

Claim 5 *The probability that x can be compressed by $\log \frac{1}{\delta}$ bits is at most δ.*

PROOF: Consider any deterministic algorithm for encoding all bit strings of length ℓ into bit strings of arbitrary length. The number of bit strings that are encoded into $\ell - b$ bits is at most $2^{\ell-b}$. So, a random bit string has probability 2^{-b} of being encoded into $\ell - b$ bits. (One can view this as a simple special case of the Kraft inequality.) □