CPSC 536N: Randomized Algorithms 2014-15 Term 2

Lecture 19

Prof. Nick Harvey *University of British Columbia*

Last time we discussed the Lovász Local Lemma. It asserts the existence of a point in a probability space that simultaneously avoids certain bad events, if their probabilities and dependences can be controlled. However, the original LLL does not give an algorithm to find such a point.

Algorithms to efficiently find such a desired point have been the subject of much research over the past few decades. Today we will discuss a remarkable algorithmic form of the local lemma by Moser from 2009.

1 An Algorithmic Local Lemma

Last time we stated the Lovász Local Lemma as follows:

Theorem 1 (The Symmetric LLL) *Suppose that there is a dependency graph of maximum degree d. If $\Pr[\mathcal{E}_i] \leq p$ for every i and $pe(d+1) \leq 1$ then $\Pr\left[\bigcap_{i=1}^n \overline{\mathcal{E}_i}\right] \geq \left(\frac{d}{d+1}\right)^n > 0$.*

Suppose we wanted to actually find an point in the probability space avoiding the bad events. If we repeatedly picked independent samples from the underlying probability distribution, the expected time to find a point avoiding the bad events could be exponential in n. Usually we would like to find this point in time $\text{poly}(n)$. This can be done for essentially all known applications of the local lemma.

Today we will discuss an algorithmic LLL which is not quite that general. Instead, we will focus on the application of the LLL to showing that SAT formulae are satisfiable, as discussed last time.

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

The theorem is stronger when α is small. The proof that we will present can be optimized to get $\alpha = 3$. The existential result from last time achieves $\alpha = \lg_2(e) \approx 1.44$, which is essentially optimal. So today's result is weaker only by a small constant factor.

The algorithm proving the theorem is extremely simple, and algorithms of this sort were certainly considered decades ago. But, they were not analyzed until 2009, when the audacious and brilliant graduate student Robin Moser at ETH made a tremendous breakthrough. Related ideas were independently discovered by Pascal Schweitzer, also a graduate student.

References: Moser's PhD Thesis Section 2.5, Schweitzer's paper.

2 The Algorithm

Moser's algorithm is given in Algorithm 1.

Algorithm 1: Algorithm to find a satisfying assignment for a k-SAT formula ϕ, under the conditions of Theorem 2.

```
Function Solve(φ):
    Set each variable in φ to either 0 or 1 randomly and independently
    while there is an unsatisfied clause C do
        Fix(C)
    Output the variable assignment
Function 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 do
        ▷ Note that possibly D = C
        Fix(D)
```

Notation. Let n be the number of variables and m be the number of clauses in ϕ. Let $T := 2^{k-\alpha}/k$ be the maximum number of occurrences of each variable. 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 $R := kT = 2^{k-\alpha}$ other clauses.

Claim 3 *Consider any call to* `Fix()` *that terminates. Let V be the set of variables that are assigned a different value after the call than before the call. Then every clause containing a variable in V is satisfied after the call.*

Proof: Consider some clause D that contains a variable in V but is not satisfied after the call terminates. Consider the last time that any variable x in D was resampled. This must have happened during some call to `Fix`(E), for some clause E that also contains x. But `Fix`(E) would not terminate until D was satisfied, which is a contradiction. □

Corollary 4 *Consider any call to* `Fix()` *that terminates. Every clause that was satisfied before the call is still satisfied after the call completes.*

Proof: If none of its variables changed, it is still satisfied. If at least one of its variables changed, Claim 3 applies. □

Corollary 5 *Assuming that C is violated, and consider any call to* `Fix(`C`)` *that terminates. The number of satisfied clauses before the call is strictly more than the number of satisfied clauses after the call.*

Proof: By Corollary 4, the number of satisfied clauses cannot decrease. Furthermore, clause C must certainly be satisfied afterwards in order for `Fix(`C`)` to terminate. □

Corollary 6 `Solve()` *calls* `Fix()` *at most m times. If the algorithm terminates, the output is a satisfying assignment.*

Proof: By Corollary 5, every call from `Solve()` to `Fix(`C`)` that terminates increases the number of satisfied clauses by at least one. The first claim follows since there are m clauses. The second claim is obvious: the only way that `Solve()` can terminate is that all clauses are simultaneously satisfied. □

So it remains to show that, with high probability, every call to `Fix()` terminates. The analysis of the algorithm is quite unusual, and counterintuitive the first time one sees it.

3 Incompressibility

Claim 7 *Let $x \in \{0,1\}^\ell$ be a uniformly random bit string of length ℓ. 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.

4 Analysis of Algorithm 1

Theorem 8 *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

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

bits, assuming that c and α 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. By Claim 7, this is possible with probability at most δ.