CS265/CME309: Randomized Algorithms and Probabilistic Analysis
Lecture #12: The Constructive Lovasz Local Lemma (Moser's Entropic Proof)

Gregory Valiant,* updated by Mary Wootters

January 25, 2025

1 Introduction

Last class we saw the statement and proof of the *existential* Lovasz Local Lemma. Today, we will see a constructive (algorithmic) version of the theorem. There has been a long progression of work towards a strong algorithmic version of the theorem over the past thirty years (e.g. [3, 2, 6, 4, 9, 7, 8, 5, 1]). In order to phrase an algorithmic version of the theorem from last class, it will be convenient to slightly restrict the set of events and probability distributions that we will consider.

Let V be a finite set of independent random variables, and let $\mathcal{A}$ denote a finite set of events that are determined by V. That is, each event $A \in \mathcal{A}$ maps the set of assignments of V to $\{0, 1\}$.

Definition 1. *Given the set of independent random variables V and set of events $\mathcal{A}$ determined by the variables of V, define the* relevant variables *for an event $A \in \mathcal{A}$, denoted $vbl(A) \subset V$ to be the smallest subset of variables that determine A. Additionally, for an event $A \in \mathcal{A}$, let $\Gamma(A) = \{B : vbl(A) \cap vbl(B) \neq \emptyset\}$ denote the set of events that share variables with A, and note that A is mutually independent from the set $\mathcal{A} \setminus \Gamma(A)$.*

The following algorithm is one extremely natural approach for finding an assignment to the variables that avoids all the events $\mathcal{A}$:

Algorithm 2. FIND ASSIGNMENT
Given V, $\mathcal{A}$:

- *Choose a random assignment σ_v for each of the random variables $v \in V$.*
- *While there exists an $A \in \mathcal{A}$ such that $A(\sigma) = 1$:*
  - *Choose (arbitrarily according to any scheme, randomized or deterministic) an event A with $A(\sigma) = 1$, and update σ by re-selecting a random assignment to the variables $vbl(A)$.*

The following theorem, due to Moser and Tardos in 2010 [8], shows that the above algorithm will, with high probability, successfully terminate quickly. The following formulation closely matches the guarantees of the "asymmetric" LLL that was mentioned in the previous lecture notes.

Theorem 1. *[8] Let V be a finite set of independent random variables. Let $\mathcal{A}$ be a finite set of events determined by the random variables in V. If there exists an assignment $x : \mathcal{A} \to (0, 1)$ such that for all $A \in \mathcal{A}$,*

$$\Pr[A] \leq x(A) \prod_{B \in \Gamma(A) \setminus \{A\}} (1 - x(B)),$$

then Algorithm 2 will find an assignment to the variables V such that no event of $\mathcal{A}$ occurs. Additionally, the expected number of "re-randomizations" is bounded by $\sum_{A \in \mathcal{A}} \frac{x(A)}{1-x(A)}$.

The above theorem implies the following algorithmic version of the simpler (symmetric) LLL:

Corollary 3. *Let V be a finite set of independent random variables. Let $\mathcal{A}$ be a finite set of events determined by the random variables in V. If for all $A \in \mathcal{A}$, $|\Gamma(A)| \leq d+1$, and $\Pr[A] \leq \frac{1}{e(d+1)}$, then Algorithm 2 will find an assignment to the variables V such that no event of $\mathcal{A}$ occurs. Additionally, the expected number of "re-randomizations" performed by the algorithm is bounded by $O(|\mathcal{A}|/(d+1))$.*

Proof. We apply Theorem 1 and set $x(A) = \frac{1}{d+1}$. To see why the assumptions of the theorem hold, note that

$$x(A) \prod_{B \in \Gamma(A) \setminus \{A\}} (1 - x(B)) \geq \frac{1}{d+1} \left(1 - \frac{1}{d+1}\right)^d \geq \frac{1}{e(d+1)}$$

where we used our assumption that $|\Gamma(A)| \leq d+1$ and the fact that $(1 - \frac{1}{d+1})^d \geq 1/e$. To conclude, recall that we assumed that $\Pr[A] \leq 1/e(d+1)$, and hence the assumptions of the theorem are satisfied. □

The original proof of Theorem 1 proceeded by bounding the expected number of times each event $A \in \mathcal{A}$ could be selected as an event whose variables are to be re-randomized (in the third line of Algorithm 2). The proof eventually turns into an analysis of a process resembling the *Galton-Watson branching process*—corresponding to the process where the "offspring" of an event A whose variables are re-randomized corresponds to the events that are must now be fixed as a result of that assignment (i.e. the events that are now true because of the new assignment to $vbl(A)$). Intuitively, as long a the expected number of offspring is < 1, this process should die out, and we should end up with an assignment s.t. no event occurs. Rather than going into this rather involved proof, we will instead describe Moser's "entropic" proof, which was not contained in the original paper.

2 Moser's Entropic Proof

The core idea of the *entropic* proof is to argue that Algorithm 2 gobbles up randomness more quickly than it actually uses it, in the sense that if the algorithm were to run for too long, then we would be able to *compress* the string of random bits used by the algorithm. And, as we show below with a simple counting argument, it is impossible to significantly compress a string of random bits. This compression/entropic argument is extremely elegant—arguing that the expected runtime must be

small because otherwise, we would be able to compress the random bits used by the algorithm. I am not aware of such an argument being used to bound the runtime of an algorithm in any other setting. There is some recent work that is trying to generalize this sort of analysis—if you are interested, see [1].

We begin with a useful fact, that one cannot compress a random string:

Fact 4 (The incompressibility of random strings)**.** *For any function f that maps t-bit binary strings to distinct strings of (possibly variable) length, if s is a uniformly random binary string of length t, then for any integer c, $\Pr[|f(s)| \leq t - c] \leq \frac{1}{2^{c-1}}$.*

Proof. Since there are 2^i strings of length i, there are at most $\sum_{i \leq t-c} 2^i < 2^{t-c+1}$ strings, that can be mapped to strings of length at most $t - c$, and hence the probability that a random length t string is in this set is at most $\frac{2^{t-c+1}}{2^t} = \frac{1}{2^{c-1}}$. □

The "entropic" proof is especially clean in the specific setting of k-SAT (rather than in the fully general LLL setting), and we will focus on k-SAT for the rest of this lecture. Consider a k-SAT formula over n variables, $x_1, \ldots, x_n$, with clauses $A_1, \ldots, A_m$, where $vbl(A_i)$ denotes the set of variables occurring in the ith clause, and $|vbl(A_i)| = k$.

Theorem 2. *There is some constant c_3 so that the following holds. Consider a k-SAT formula with m clauses over n variables. If, for each clause C in the formula, there are at most $d = 2^{k-c_3}$ clauses whose variable sets intersect the variables in clause C, then the formula is satisfiable.*

Moreover, there is a randomized algorithm that starts with any fixed assignment, iteratively re-randomizes the assignment to variables in some unsatisfied clause, and with probability at least 0.9 produces a satisfying assignment after at most $O(m \log m)$ re-randomizations and in time polynomial in n and m.

Proof. Ultimately, we will argue that if the algorithm were to run for too long, in expectation, then we would end up a with a protocol that compresses the random bits used by the algorithm. To do this, let's consider a variant of Algorithm 2:

Algorithm 5. FIND ASSIGNMENT AND PRINT STUFF

Given φ:

- *Choose a random assignment σ for each of the variables that appear in φ.*
- *For each clause C_i in φ that is violated by σ:*
  - *print "Running* FIX *on Clause i"*
  - *Define a global counter t, initialized to zero.*
  - *Define a global iteration limit T, to be defined in the proof.*
  - FIX(φ, i, σ) *(See Alg. 6 below).*
- *Return σ.*

Algorithm 6. FIX(φ, i, σ)

Given a formula φ, a clause index i, an assignment σ:

- *If $t = T$: (recall that t and T are global variables defined in Alg. 5).*
  - *print "Reached iteration limit, and the current assignment is σ"*
  - *halt and return* FAIL.
- $t \leftarrow t + 1$
- *Flip k random coins, $r_1, \ldots, r_k \in \{0,1\}$.*
- *Update σ by setting the j'th variable that appears in the clause C_i to r_j. (For simplicity assume that exactly k variables appear in each clause).*
- *Suppose that $C_{i_1}, C_{i_2}, \ldots, C_{i_{d+1}}$ are the clauses in φ that share variables with C. (Including C itself).*
- *For $\ell = 1, 2, \ldots, d+1$:*
  - *If C_{i_ℓ} is violated:*
    - *print "Trying to fix the ℓ'th child..."*
    - FIX$(\varphi, i_\ell, \sigma)$
- *print "All done, moving back up a level."*

You can check that Algorithm 5 above is essentially the same as Algorithm 2, except for a few changes:

- It's written as a recursive, rather than iterative, algorithm.
- We've fixed the order that it corrects violated clauses in,[1]
- There's an iteration cap T. If we call FIX T times without finishing, the algorithm will halt and return FAIL.
- The algorithm makes some print statements.

You might recognize the print statements in FIX (Algorithm 6) as the sort of statements you'd lazily write when trying to debug a recursive algorithm.[2] Given the output from these print statements, you can recover the execution path of the algorithm. When debugging, you could use this information to find out what went wrong. In our case, we will use this output as a method of *compressing* the random bits flipped by the algorithm. In more detail, you can view this algorithm in two ways:

1. This algorithm takes as input φ, flips some random coins, makes some print statements for some reason, and eventually returns σ or FAIL
2. This algorithm takes as input a long string of random bits, has some internal state given by φ, possibly computes σ for some reason, and then outputs a series of print statements.

[1] The algorithm does work no matter what order you correct the clauses in, but it will be easier to analyze if we fix this order, so let's do that.

[2] If you don't recognize them as this, you are a better programmer than I am...

We will show that the second way of viewing this algorithm is in fact a compression algorithm for random bits. If the algorithm runs for too long, then the compression algorithm will be too good, violating Fact 4. This will be a contradiction, implying that the algorithm cannot run for too long.

Claim 7. *Consider a single call to* FIX *in Algorithm 5 (say, the first one). If* FIX *runs for T timesteps and outputs* FAIL*, then it is possible to recover all of the random coins flipped by* FIX*, as well as the original assignment σ that was chosen in Algorithm 5, from the print statements.*

Before we prove Claim 7, let's see why it implies the theorem.

Consider running Algorithm 5 with the iteration limit T set to $T = \infty$. First, observe that the running time is at most the number of clauses times the maximum running time of each call to FIX, and that if Algorithm 5 successfully finishes, it will return a satisfying assignment σ. This is because each time Algorithm 5 calls FIX and successfully finishes it, it will fix the clause that it was called on, and won't break any other clauses. (It might accidentally fix another broken clause, but that only helps us).

Thus, we want to show that the running time of FIX, even with $T = \infty$, is small. To do that, we'll analyze the running time with

$$T = \frac{2\log m + c_2}{k - log(d+1) - c_1}$$

for some constant c_1 and c_2 that will be determined below, and show that if FIX is likely to still be running after T re-randomizations, then we'll have a compression algorithm that is too good. This will be a contradiction, and we'll conclude that FIX is very likely to finish on its own before making T re-randomizations.

Notice that as long as $d + 1 \leq 2^{k-c_1-1}$, we have $T = O(\log m)$.

So fix T as above, and suppose that FIX runs for T re-randomizations and then outputs FAIL. How many bits of information does the output along the way? We have:

- `"Running Fix on Clause i"` before we call FIX. This is $\log m + O(1)$ bits. The $\log m$ comes from needing to write down i, which could be one of m things.
- `"Reached iteration limit, and the current assignment is σ"` when we hit the iteration limit. This is $n + O(1)$ bits, because we need n bits to write down σ.
- For at most T iterations (one for nearly every call of FIX), we print `"Trying to fix the ℓ'th child..."`. This is $T(\log(d+1) + O(1))$ bits, since we need $\log(d+1)$ bits to print ℓ, which could be one of $d + 1$ things.
- For at most T iterations, we print `"All done, moving back up a level."` This is $O(T)$ bits.

Altogether, we print at most

$$\log m + n + (c_1 + \log(d+1))T$$

bits, for some constant c_1 (assuming $T \geq 1$).

On the other hand, by Claim 7, given the output from these print statements, we are able to recover the random coins (the r_i's) that were flipped by the algorithm, as well as the original assignment σ. The number of random bits here is $n + Tk$: there are n random bits in σ, and Tk flipped during the T re-randomizations.

Suppose towards a contradiction that this run of FIX runs for T re-randomizations with probability greater than $1/(10m)$. (And with probability at most than $1/(10m)$, it stops before then on its own).

Then, with probability strictly less than $1 - 1/(10m)$, the algorithm transmits $\log m + n + c_1 T$ bits, and this can be used to recover $Tk + n$ bits of randomness. This contradicts Fact 4 (choosing $c \leftarrow 1 + \log(10m)$ in the statement of the Fact), provided that

$$\log m + n + (c_1 + \log(d+1))T < n + Tk - \log(10m) - 1,$$

which is true by our choice of

$$T = \frac{2\log m + c_2}{k - log(d+1) - c_1}$$

for an appropriate choice of the constant c_2.

We conclude that the probability that a single run of FIX going for T re-randomizations without returning is at most $1/(10m)$. We can then union bound over all m times that FIX is called in Algorithm 5 and conclude that with probability at least 0.9, Algorithm 5 returns a satisfying assignment for φ, and none of the calls to FIX took longer than T iterations. With our assumption on d (the number of clauses that share variables with any other clause), the number of re-randomizations is thus at most $O(m \log m)$, and the total running time is polynomial in n and m.

It remains to prove the claim.

Proof of Claim 7. We need to show that, given the output of the print statements, we can recover all of the random coin flips from the re-randomizations, as well as the original assignment σ.

To see this, we can work backwards through the algorithm. The trail of print statements will allow us to reconstruct the recursion tree, and in particular we know which clause was being re-randomized in each call.

Starting at the very end, we know that the algorithm ended up with some final assignment σ (which we know because it printed it out), and we also know the identity of the last clause that it re-randomized; let's say that it was C_j, and that the variables in C_j are $x_{j_1}, \ldots, x_{j_k}$. But since there's only one way to assign values to $x_{j_1}, \ldots, x_{j_k}$ that would result in C_j being violated, we know exactly what the assignment σ was before we did the last re-randomization, and we know what the last k randomizing bits were. Now we can work backwards, repeating this logic until we get all the way back to the starting string σ. □

This completes the proof of the theorem. □

2.1 Discussion

One curious punchline that emerges from the proof is that we *want* the algorithm to use lots of randomness. If, instead of using k bits of randomness with every step, it only used $k/2$ bits, then the proof would not work, and we would not end up with any bound on the runtime. For example, consider a "greedy" algorithm which, rather than re-randomizing the variables in a given clause, tries to find an assignment to those k variables that satisfies that clause and as many other clauses are possible. Such a greedy scheme slightly reduces the amount of randomness consumed by the algorithm, and hence the proof from above would result in a worse bound on the expected runtime (or a possibly infinite bound). In practice, for some classes of formula, people have observed that

this sort of greedy algorithm is actually much worse than actually re-randomizing. (At first, the greedy algorithm seems to make great progress, but then things start to stagnate/loop.) The above proof offers one conceptual explanation for why, in these settings, we want to maximize the amount of randomness the algorithm is actually using.

References

[1] Dimitris Achlioptas and Fotis Iliopoulos. Random walks that find perfect objects and the lovász local lemma. *Journal of the ACM (JACM)*, 63(3):1–29, 2016.

[2] Noga Alon. A parallel algorithmic version of the local lemma. *Random Structures & Algorithms*, 2(4):367–378, 1991.

[3] József Beck. An algorithmic approach to the lovász local lemma. i. *Random Structures & Algorithms*, 2(4):343–365, 1991.

[4] Artur Czumaj and Christian Scheideler. Coloring nonuniform hypergraphs: A new algorithmic approach to the general lovász local lemma. *Random Structures & Algorithms*, 17(3-4):213–237, 2000.

[5] Bernhard Haeupler, Barna Saha, and Aravind Srinivasan. New constructive aspects of the lovász local lemma. *Journal of the ACM (JACM)*, 58(6):1–28, 2011.

[6] Michael Molloy and Bruce Reed. Further algorithmic aspects of the local lemma. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 524–529, 1998.

[7] Robin A Moser. A constructive proof of the lovász local lemma. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 343–350, 2009.

[8] Robin A Moser and Gábor Tardos. A constructive proof of the general lovász local lemma. *Journal of the ACM (JACM)*, 57(2):1–15, 2010.

[9] Aravind Srinivasan. Improved algorithmic versions of the lovász local lemma. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 611–620. Citeseer, 2008.