CS 761: Randomized Algorithms **University of Waterloo: Winter 2025**

Lecture 8

February 6, 2025

Instructor: Sepehr Assadi

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

Topics of this Lecture

1 Proof of Lovász Local Lemma

In the previous lecture, we saw the Lovász Local Lemma (LLL) and its amazing power. We now prove a slightly weaker form of LLL to provide more intuition about it (notice that we effectively replace $1/e$ in the original statement with a weaker constant of $1/4$).

Theorem 1 (**A "Weak" Form of LLL**). *Suppose $B_1, \ldots, B_n$ are a collection of events. If:*

1. $\Pr(B_i) \leqslant p$ *for every* $i \in [n]$, *for some* $p \in (0,1)$;
2. *and, the events admit a dependency graph with maximum degree* $d \geqslant 1$, *i.e., for each event* B_i, *there exists a set* $N(i)$ *of size at most* d *such that for all* $T \subseteq [n] \setminus (N(i) \cup \{i\})$,

$$\Pr(B_i \mid \wedge_{j \in T} B_j) = \Pr(B_i).$$

Then, as long as

$$p \cdot d \leqslant 1/4$$

the probability that ***none of*** $B_1, \ldots, B_n$ *happens is strictly more than zero.*

Proof. Define A_i to be the complement of the event B_i. Moreover, for any $i \in [n]$, define $A_{<i}$ as the event that $A_1, \ldots, A_{i-1}$ all happen. Our goal is to prove that

$$0 < \Pr\left(\wedge_{i=1}^{n} \bar{B}_i\right) = \Pr\left(\wedge_{i=1}^{n} A_i\right) = \prod_{i=1}^{n} \Pr\left(A_i \mid \wedge_{j=1}^{i-1} A_j\right) = \prod_{i=1}^{n} \Pr(A_i \mid A_{<i}).$$

Thus, we have to prove that

$$\Pr(A_i \mid A_{<i}) > 0 \iff \Pr(B_i \mid A_{<i}) < 1$$

for all $i \in [n]$. We actually prove a stronger statement inductively:

Induction hypothesis: *For any $i \in [n]$ and any set $S \subseteq [n] \setminus \{i\}$,*

$$\Pr\left(B_i \mid A_S\right) \leqslant 2p,$$

where A_S is defined as $\wedge_{j \in S} A_j$.

The base case for each $i \in [n]$ and $S = \emptyset$ follows immediately because $\Pr\left(B_i\right) \leqslant p$ by the theorem statement. We now prove the induction step.

Step 1. We know that B_i only depends on at most d other events in $N(i)$ so we should find a way to "get rid of" the remaining terms in A_S. To do so, we write,

$$\begin{aligned}
\Pr\left(B_i \mid A_S\right) &= \Pr\left(B_i \mid A_{S \cap N(i)}\ A_{S \setminus N(i)}\right) \\
&= \frac{\Pr\left(B_i \wedge A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right)}{\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right)} && \text{(by the definition of conditional probability)} \\
&\leqslant \frac{\Pr\left(B_i \mid A_{S \setminus N(i)}\right)}{\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right)} && \text{(as } \Pr(C \wedge D) \leqslant \Pr(C) \text{ for any events } C, D\text{)} \\
&= \frac{\Pr\left(B_i\right)}{\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right)} && \text{(because } B_i \text{ is independent of events outside } N(i)\text{)} \\
&\leqslant \frac{p}{\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right)}. && \text{(as } \Pr\left(B_i\right) \leqslant p \text{ in the theorem statement)}
\end{aligned}$$

The only "real" inequality above is in dropping $\wedge A_{S \setminus N(i)}$ (the other inequality might as well be tight also because we have no control over the gap between $\Pr(B_i)$ and p in the theorem statement). As we shall see, the "math is going to work out" even when taking this inequality but it is good to see some intuition why this is the case. This is because, $A_{S \cap N(i)}$ only contains d terms and in the next step we are going to prove that these terms actually happen with a "large enough" probability (a constant more than zero); As a result, we are *not* "dropping" a very low probability event that can make the inequality quite loose.

Step 2. We know need to lower bound the denominator of the RHS above. But now, this term only depends on d events in total and we can try to simply use a *union bound* to get a loose bound here. Specifically,

$$\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right) = 1 - \Pr\left(\vee_{j \in S \cap N(i)} B_j \mid A_{S \setminus N(i)}\right) \geqslant 1 - \sum_{j \in S \cap N(i)} \Pr\left(B_j \mid A_{S \setminus N(i)}\right).$$

Given that $N(i)$ has at least one element (as otherwise B_i is independent of all other events and trivially satisfies the induction hypothesis), we have that $|S \setminus N(i)| < |S|$. Thus, we can apply our induction hypothesis and obtain that for every $j \in S \cap N(i)$,

$$\Pr\left(B_j \mid A_{S \setminus N(i)}\right) \leqslant 2p.$$

Plugging this bound above gives us

$$\Pr\left(A_{S \cap N(i)} \mid A_{S \setminus N(i)}\right) \geqslant 1 - \sum_{j \in S \cap N(i)} 2p \geqslant 1 - 2p \cdot d \geqslant 1/2,$$

where the last inequality is by the assumption in the theorem statement that $p \cdot d \leqslant 1/2$.

Plugging in the bounds of step 2 on the last equation of step 1 give us

$$\Pr\left(B_i \mid A_S\right) \leqslant \frac{p}{1/2} \leqslant 2p,$$

as desired. This concludes the proof. □

It is worth mentioning that the statement of Theorem 1 is actually the original version of LLL proven by Erdős and Lovász in [EL75].

Another application of LLL: "Sparse" CNFs. Let us use Theorem 1 to showcase yet another application of LLL in preparation for the main topic of this lecture that will appear afterwards.

Recall that for any integer $k \geqslant 1$, a k-CNF (conjunctive normal form) is a set of m *clauses* each consisting of 'OR' of k *literals* over n variables; moreover, these clauses are 'AND' together. E.g., the following is an example of a 3-CNF:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_2 \vee \neg x_4) \wedge (x_4 \vee x_6 \vee x_7) \wedge \cdots$$

We are going to prove that if in a k-CNF, every variable appears in "few" other clauses; then, the CNF is always satisfiable, i.e., there exists an assignment to the n variables such that every clause becomes true.

Proposition 2. *For any $k \geqslant 2$, let Φ be a k-CNF such that every variable appears in at most $2^k/4k$ other clauses. Then, Φ is satisfiable.*

Proof. The proof is again a simple application of LLL. Suppose we pick the assignment to $x_1, \ldots, x_n$ randomly from $\{0,1\}^n$. Define the 'bad' event B_i for each clause $i \in [m]$ as the event that under this assignment, the i-th clause is not satisfied. Given each clause is 'OR' of k literals, there exists exactly one assignment to its k variables that does not satisfy the clause. Thus,

$$\Pr(B_i) = 2^{-k}.$$

On the other hand, each bad even B_i is entirely independent of all bad events B_j that their clauses to do not share any variable with those of B_i. Since each variable in clause i can belong to at most $2^k/8k$ other clauses and there are k variables in the clause i, B_i is independent of all but at most

$$d = \frac{2^k}{4k} \cdot k = \frac{2^k}{4}$$

other bad events. As

$$2^{-k} \cdot (\frac{2^k}{4}) \leqslant \frac{1}{4},$$

we can apply LLL in Theorem 1 and obtain that with non-zero probability, none of the bad events happen. This means that there is a satisfying assignment for Φ, concluding the proof. □

2 An *Algorithmic* LLL?

Up until this point, we only used LLL to prove *existence* of objects. But, what if we would like to *find* those objects as well?

For instance, in Proposition 2, can we also *find* the satisfying assignment? LLL implies that the probability of the bad events not happening is non-zero, so if we continue sampling random assignments, we will eventually find a satisfying assignment. But, using the probabilities implied by LLL only implies that the probability of finding a satisfying assignment is $> 2^{-n}$. But that requires running the algorithm roughly 2^n times before finding the 'right' assignment. This is basically the same as (in fact, even worse than) enumerating all assignments and finding a satisfying one (that LLL guarantees its existence). But, can we do this in polynomial time?

This is the content of *algorithmic* Lovász Local Lemma, a beautiful area of research that has been developed over a decade ago. We will see an application of this to finding an assignment in Proposition 2 in polynomial time due to a brilliant idea of Moser [Mos09].

2.1 An Algorithmic Version of Proposition 2

The algorithm is as follows.

Algorithm 1.

1. Let $C_1, \ldots, C_m$ be the clauses and x be a random assignment in $\{0,1\}^n$ to the variables.
2. For $i = 1$ to m: if clause C_i is violated under x, run the subroutine $\texttt{Fix}(C_i)$.

Subroutine $\texttt{Fix}(C_i)$:

1. *Resample* the variables in the clause C_i from $\{0,1\}^k$.
2. Go over 'neighbors' of C_i, namely, clauses that share a variable with C_i—including C_i itself—denoted by $N(C_i)$, and for each clause $C \in N(C_i)$, if C is violated now, recursively call $\texttt{Fix}(C_i)$.

Note that it is not clear apriori that this algorithm ever terminates. This is because we maybe "fixing" one clause, but then create many violated clause as a result, and now have to fix them and just continue doing this in a loop forever! Nevertheless, we can at least say that *if* the algorithm terminates, then the resulting assignment is feasible.

Lemma 3. *If Algorithm 1 terminates, then the final assignment x to the variables satisfy all clauses.*

Proof. We inductively prove that after iteration $i \in [m]$ of the for-loop, all clauses $C_1, \ldots, C_i$ are satisfied. This is vacuously true for $i = 0$.

Now consider some iteration $i \geqslant 1$. By induction hypothesis, $C_1, \ldots, C_{i-1}$ are already satisfied so we only need to worry about C_i. If C_i is already satisfied, then we are done, so suppose C_i is not satisfied. But, then we will be calling $\texttt{Fix}(C_i)$ which "fixes" C_i; in addition, if any of the neighbors of C_i become violated, we recurse on them also, so this chain of recursion never terminates before making sure everything that was satisfied before the call $\texttt{Fix}(C_i)$ is also satisfied now. This means that *if* the call to $\texttt{Fix}(C_i)$ terminates, then, $C_1, \ldots, C_{i-1}$ remain satisfied, and C_i is also now additionally satisfied. This proves the induction.

Hence, after the for-loop finishes (if ever), all clauses are satisfied. □

Thus, it "only" remains to prove that this algorithm actually terminates. What makes this hard to argue is that seemingly no particular "progress" is being made here; we may make a single clause satisfied when calling `Fix` and in the process violate several other clauses, hence, making the matter worse in some sense! This is where Moser's brilliant idea comes into picture that provides a *unique* way of analyzing these types of processes, which is called the *entropy compression* method, as termed by Terry Tao.

2.2 Entropy Compression Method and Runtime Analysis of Algorithm 1

To continue, we just need the following simple information-theoretic lemma.

Lemma 4. *Let $\delta \in (0,1)$ and $f : \{0,1\}^m \to \{0,1\}^t$ be any fixed function with the following property: if we sample x uniformly at random from $\{0,1\}^m$, we can recover x from $f(x)$ with probability at least δ. Then, we should have $t \geqslant m - \log(1/\delta)$*

This lemma says that if we attempt to "compress" the strings in $\{0,1\}^m$ to shorter lengths, most of the times we will not be able to recover the strings correctly anymore. Even more informally speaking, *we cannot hope to compress random strings.*

Proof. Let $X \subseteq \{0,1\}^m$ be the strings that can be recovered uniquely from $f(x)$. The function f from $X \to \{0,1\}^t$ should be injective as otherwise we cannot recover x uniquely from $f(x)$ if $f(y) = x$ also and both $x, y \in X$. This means that $|X| \leqslant 2^t$. But, we also have $|X| \geqslant \delta \cdot 2^m$ as a uniformly chosen $x \in \{0,1\}^m$ can be recovered from $f(x)$ with probability at least δ. Hence,

$$2^t \geqslant |X| \geqslant \delta \cdot 2^m \implies t \geqslant m - \log(1/\delta),$$

concluding the proof. □

We are going to prove that Algorithm 1 is "trying" to compress random bits—thus, if it gets to run for a very long time with a large probability, it will manage to compress random bits beyond what is allowed by Lemma 4, a contradiction. In particular, we will prove the following main theorem.

Theorem 5 ([Mos09])**.** *For any $k \geqslant 2$, let Φ be a k-CNF such that every variable appears in at most $2^{k-c}/k$ other clauses for some large constant $c > 10$. Then, Algorithm 1 finds a satisfying assignment of Φ in $O(m \cdot k)$ time with probability at least 0.99.*

Let s be a parameter to be determined later (we fill set $s = m + O(1)$ eventually). Consider the following modification to the algorithm (this is only for the purpose of the analysis): we only allow the algorithm to call the subroutine `Fix` at most s times and after that we simply terminate the algorithm. Our goal is to show that with high constant probability, the algorithm actually does not get to make these s calls and thus finishes beforehand with the answer.

Firstly, how many random bits the modified algorithms generates throughout the whole process? Well, it starts with n random bits at the beginning and then each call to `Fix` tosses k new random bits so the total number of bits is $n + s \cdot k$. We are going to assume that the algorithm simply tosses all these coins in advance. Consider the following way of "compressing" these random bits.

The compression scheme. We are going to maintain a *log* of what the algorithm does so that we can regenerate all its decision from this log. In the following, let $R := 2^{k-c}$ be the number of neighbors of any single clause. We do as follows.

- Firstly, we will write a string of m bits where its i-th bit is 1 if in the i-th for-loop of Algorithm 1, the algorithm had to call `Fix`(C_i) and is 0 otherwise.

  Notice that given this m bit string, we can figure out exactly what were the "top level" calls to `Fix` (although of course not the "inner" recursive calls).

- We then follow these m bits with the following strings. Consider the first time `Fix` was called, say, on a clause C. Write a $(\log R)$-bit string to name which neighbor of C is being called next in `Fix`. We continue doing this but also whenever a call to `Fix` is terminated, we will write a 'terminated' symbol in $O(1)$ bits.

  For instance, suppose the algorithm calls `Fix`(100) and then calls `Fix`(2) (namely, as in, the second neighbor of 100), and terminates, and then call `Fix`(5) and inside it call `Fix`(7) also and so on; then, we will write

  $$100, 2, \text{'terminated'}, 5, 7, \cdots$$

  Notice that each of these numbers can be written with $\log R + O(1)$ bits. In particular, if the algorithm makes s calls to `Fix`, we can write this part using

  $$s \cdot (\log R + O(1))$$

  bits.

- Finally, we will write the assignment to the variables after the s calls in n bits. (If the algorithm finishes before making s call, we let the log to be simply the empty string; in other words, the log is only used when the algorithm finishes all s calls to `Fix` without finding the assignment and thus terminates instead).

We claim that using this information, we can recover *all* the random bits used by the algorithm exactly. This is because, we can look at the very final call to `Fix` (which can be recovered from the log). The only reason the algorithm called this clause C is because the current assignment of the variables did not satisfy this clause. But, there is only one assignment of the k variables that violates this clause. So, we can recover

what was the state of assignment x when this call to `Fix` happened. We can then backtrack this way and now what were the values of x in every step and since we know the clauses also, we know exactly what are the random bits used.

Now, suppose that the original algorithm does not terminate with probability at least δ. This in turn means that the modified algorithm gets to output a complete log with probability at least δ, from which, we can recover all the $n + s \cdot k$ random bits. By Lemma 4, this means the length of the log should be at least

$$n + s \cdot k - \log(1/\delta).$$

But, on the other hand, the length of the log is also at most

$$m + s \cdot (\log R + O(1)) + n$$

bits by construction. Recall that $\log R = k - c$ for some arbitrarily large constant $c > 1$, and thus, we have

$$m + s \cdot (\log R + O(1)) + n = m + s \cdot (k - c + O(1)) + n \geqslant n + s \cdot k - \log(1/\delta).$$

By taking c to be large enough to become equal to the $O(1)$-term plus one, we get

$$m + s \cdot k - s \geqslant s \cdot k - \log(1/\delta) \implies s \leqslant m + \log(1/\delta).$$

This implies that for any $\delta \in (0, 1)$, the algorithm terminates after at most $m + \log(1/\delta)$ calls to `Fix`, concluding the proof of Theorem 5.

References

[EL75] Paul Erdős and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and finite sets*, 10(2):609–627, 1975. 2

[Mos09] Robin A. Moser. A constructive proof of the lovász local lemma. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 343–350. ACM, 2009. 3, 5