6.5420 Randomness and Computation February 10, 2026

Lecture 3: The Lovasz Local Lemma

Lecturer: Ronitt Rubinfeld *Scribe: Joseph Othman*

1 Introduction

We want another way of showing that it is possible for no bad events to happen. Let our bad events be $A_1, A_2, \cdots A_n$. Before, we used the union bound:

$$\mathbb{P}[\text{any } A_i \text{ occurs}] \leq \sum_{i=1}^{n} \mathbb{P}[A_i] \leq n \cdot p < 1$$

as long as $\mathbb{P}[A_i] \leq p < \frac{1}{n}$ for all i. This is not very sharp in general, and does not take advantage of any dependence/independence that we might know about the A_i's.

Instead, if we had full independence, then we could say

$$\mathbb{P}[\text{no } A_i \text{ occurs}] = \prod_{i=1}^{n} \mathbb{P}[A_i \text{ doesn't occur}] > 0$$

which is true as long as none of the $\mathbb{P}[A_i]$'s are 1. However, this requires full independence. We want a middle ground between the union bound and this case: a decent bound that takes advantage of whatever independence structure we have.

Definition 1. *An event A is independent of events $B_1, \cdots, B_k$ if $\forall J \subset [k]$ we have*

$$\mathbb{P}\left[A \cap \bigcap_{j \in J} B_j\right] = \mathbb{P}[A] \cdot \mathbb{P}\left[\bigcap_{j \in J} B_j\right]$$

Note that we cannot just say that A is independent of B_1 and A is independent of B_2 and so on; we must let J range over all sets of elements of the other events.

Definition 2. *Let $A_1, \cdots, A_n$ be some events. We define a dependency digraph of these events to be the graph $D = (V, E)$ with $V = [n]$ where A_i is independent of all A_j such that $i \neq j$ and $(i, j) \notin E$.*

We now state the Lovasz Local Lemma.

Theorem 3 (Symmetric Lovasz Local Lemma)**.** *Let $A_1, A_2, \cdots, A_n$ be events such that $\mathbb{P}[A_i] \leq p$ for all i. Say these events have dependency digraph D such that D has max degree at most d. Then*

$$epd \leq 1 \implies \mathbb{P}\left[\bigcap_{i=1}^{n} \overline{A_i}\right] > 0$$

Let's compare to the union bound. Above, we said that we need $n \cdot p < 1$ in order to successfully show that no events occur. But with the LLL, if we have $d \leq 4$ (as an example), then we only need $p \leq \frac{1}{4e}$ to show that no bad event occurs in some case. Note in this case that p does not depend on the number of events, though generally it will when d has some (hopefully slow) dependency on n.

2 First Example: Monochromatic Sets

Theorem 4. *Say we have sets $S_1, \cdots, S_m \subset X$ where each $|S_i| = l$ and $|X| = n$. Say that each S_i intersects at most d other sets. If*

$$ed \leq 2^{l-1}$$

then we can $2-$color X such that no S_i is monochromatic.

(Last time, we needed $m \leq 2^{l-1}$ instead, but now m is allowed to be large given our degree bound).

Proof. Color each element of X either red or blue, i.i.d with probability $\frac{1}{2}$. Let A_i be the event that S_i is monochromatic; note that $\mathbb{P}[A_i] = \frac{1}{2^{l-1}}$.

Note that A_i is independent of all A_j such that $S_i \cap S_j = \varnothing$. Then each event depends on at most d others, and thus the Lovasz Local Lemma implies that since $e \cdot \frac{1}{2^{l-1}} \cdot d \leq 1$, then with probability > 0 we have that none of the A_i occur, and thus some coloring achieves no monochromatic set as desired. □

3 Second Example: Boolean Formula

Theorem 5. *Given a CNF formula such that we have l variables in each clause, and each clause intersects at most d others, if we have*

$$\frac{ed}{2^l} < 1$$

then there is a satisfying assignment of the formula.

Example 6. *This is a CNF formula*

$$(X_1 \vee X_2 \vee X_3) \wedge (\bar{X_1} \vee \bar{X_2} \vee X_5) \wedge (X_4 \vee X_8 \vee X_9) \wedge (X_1 \vee X_6 \vee X_7)$$

There are 4 clauses; the first, second, and fourth clauses all intersect each other (all contain X_1 or $\bar{X_i}$), and none intersect the third clause.

We will let n be the number of variables, and m be the number of clauses. The proof of the existence is the same as the first example.

We now try to construct a satisfying assignment. Note that the Symmetric LLL we have described above, does not tell us how to construct an algorithm.

Theorem 7. *Say we have a CNF formula $\varphi = \bigwedge_{i=1}^m C_i$ where C_i is a clause with l literals and each C_i intersects with at most d other clauses. If*

$$d + 1 \leq \frac{2^l}{2^{2.1}}$$

then we can find an assignment of X (the variables) such that each C_i is satisfied in polynomial time with respect to $m, d, n = |X|$

We will use Moser's algorithm.

3.1 Moser's Algorithm for Boolean Formulas

Here are the steps of Moser's algorithm:

1. Pick a random assignment of $x_1, \cdots, x_n$.
2. For each clause $i \in [m]$, if C_i is unsatisfied, do Fix(C_i) ($*$).
3. Output the assignments.

We need to define Fix(C):

1. Rerandomize variables in C.
2. For C' in $\{C\} \cup \{\text{neighbors of } C\}$, if C' is unsatisfied, do Fix(C').

If the algorithm terminates, then we have a satisfying assignment. To explain why it terminates, we will view Moser's algorithm as a "compression algorithm" whose input is a random string R.

3.2 Compression Algorithm

We will say that the CNF φ is fixed.

Input: to be a random string R such that $|R| = t$.
Rule: if Moser's algorithm terminates or runs out of bits in R, stop. When we stop, output E, the encoding of the trace of computation.

3.2.1 Trace of Computation

We need to define what the trace of computation has. This is described here.

- bit string b_i for $i \in [m]$. $b_i = \begin{cases} 1 & \text{Fix}(C_i) \text{ called on line } (*) \\ 0 & \text{otherwise} \end{cases}$
- for each recursive call to Fix, record:
  1. which neighbor clause called
  2. bits for recursive structure:
     - $b_1 = \begin{cases} 1 & \exists \text{ child call} \\ 0 & \text{otherwise} \end{cases}$
     - $b_2 = \begin{cases} 1 & \exists \text{ subsequent sibling call in tree} \\ 0 & \text{otherwise} \end{cases}$
- Final variable assignment.
- Any remaining bits in R.

Noteably, none of the random bits that were actually used in Moser's algorithm are not part of the trace of computation. Only the bits of R that were not used are in this trace.

For this example, we will use 1 and true interchangeably, as well as 0 and false.

Example 8. *Let* $R = 000001100\ 110\ 000\ 101\ 110\ 11001$*. Recall our clause*

$$(X_1 \vee X_2 \vee X_3) \wedge (\bar{X_1} \vee \bar{X_2} \vee X_5) \wedge (X_4 \vee X_8 \vee X_9) \wedge (X_1 \vee X_6 \vee X_7)$$

The first 9 *bits (*000001100*) are our random assignment of the variables. Then initially our clause looks like this:*

$$(F \vee F \vee F) \wedge (T \vee T \vee F) \wedge (F \vee F \vee F) \wedge (F \vee T \vee T)$$

Then Moser's algorithm will call Fix(1)*, as the first clause is not satisfied. We will use the next* $l = 3$ *bits of* R *(*110*) to substitute into* X_1*,* X_2*,* X_3 *and so we take* $X_1 = T, X_2 = T, X_3 = F$*. Then our clause becomes*

$$(T \vee T \vee F) \wedge (F \vee F \vee F) \wedge (F \vee F \vee F) \wedge (T \vee T \vee T)$$

note that the only variables that have changed are X_1*,* X_2*, and* X_3*.*
We can use an adjacency list representation of the clauses to now check when to call Fix *again. Note that our adjacency list is the following:*

$$\begin{aligned} C_1 &: [C_1, C_2, C_4] \\ C_2 &: [C_1, C_2, C_4] \\ C_3 &: [C_3] \\ C_4 &: [C_1, C_2, C_4] \end{aligned}$$

For our encoding, we will replace the names of the clauses using lexicographical number of the lists. Shown below are the names of the entries of the adjacency lists.

$$\begin{aligned} C_1 &: [00, 01, 10] \\ C_2 &: [00, 01, 10] \\ C_3 &: [00] \\ C_4 &: [00, 01, 10] \end{aligned}$$

Clause 1 *might have to call another* Fix*. We don't have to call* Fix(1)*, but now we must (recursively) call* Fix(2)*. Our next bits are* FFF*, so we will take* $X_1 = F, X_2 = F, X_5 = F$*. This gives us the clause*

$$(F \vee F \vee F) \wedge (T \vee T \vee F) \wedge (F \vee F \vee F) \wedge (F \vee T \vee T)$$

Checking the adjacency list of C_2*, we have messed up* C_1*, and thus we must call* C_1*. Our next bits are* 101*, so we take* $X_1 = T$*,* $X_2 = F$*,* $X_3 = T$*, and thus we get the clause*

$$(T \vee F \vee T) \wedge (F \vee T \vee F) \wedge (F \vee F \vee F) \wedge (T \vee T \vee T)$$

Note that in our encoding, we currently know that the first entry of b *is* 1*. From this bit we can draw this tree:* $(1) \to (01, 1, 0) \to (00, 0, 0)$*. This tree describes the recursive calls of* Fix *in the following manner: the first bit is represented by the (*1*). The child that calls fix is named* 01 *(since clause* 2 *being unsatisfied makes our algorithm call* Fix(2)*). This leads to the child call* Fix(1)*, which is named* 00*. This makes no other calls, and thus has no child calls, so the third bit is* 0*. Traversing up, we go back to clause* 2 *and don't have to make any subsequent calls, by checking the rest of the adjacency list of the caller and checking that all clauses are satisfied, and thus the sibling bit here is* 0*. Then we can fill out the child and sibling bits as* $(1, 0)$ *for the call to* Fix(2) *and* $(0, 0)$ *for the call to the nested* Fix(1)*.*

We return to the algorithm; we check the adjacency list of C_1 *and see that all clauses are satisfied. Then we go to clause* 2*, which is satisfied, so we don't call* Fix *here either. In the encoding, we get* *that the second entry of* b *is* 0 *since we did not call* Fix*. Then we go to clause* 3*, which is unsatisfied, so we call* Fix(3)*. Our next bits of* R *are* 110*, which gives us* $X_4 = T, X_8 = T, X_9 = F$*. This gives us the clause*

$$(T \vee F \vee T) \wedge (F \vee T \vee F) \wedge (T \vee T \vee F) \wedge (T \vee T \vee T)$$

In the encoding, we have called Fix(3) *in (*$*$*), so we have that the third bit is* 1*. Moreover, we know that the assignment worked, which gives us the tree* $(1) \to (0)$ *in the tree structure. Finally, we go to clause* 4 *and see that it is satisfied, which gives us that the last bit of* b *is* 0*. Then our encoding* E *becomes:* 1010 1 0110 0000 0 101101110 11001*. The first section is* b*. The next is the tree from bit* 1*. The next is the tree from bit* 3*. The next is the final variable settings, and then the last section is the unused bits of* R*.*