CS 466/666: Algorithm Design and Analysis **University of Waterloo: Fall 2023**

Lecture 21

November 22, 2023

Instructor: Sepehr Assadi

Topics of this Lecture

1 Recap of Lovász Local Lemma

In the previous lecture, we saw the following Lovász Local Lemma (LLL):

Theorem 1 (**Symmetric 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 admits a dependency graph with maximum degree* $d \geqslant 1$,

Then, as long as

$$e \cdot p \cdot (d+1) \leqslant 1 or, \qquad (e \sim 2.73 \cdots \text{ is the natural number})$$

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

We then saw that we can use this extremely strong tool to prove *existence* of various objects using the probabilistic method (among other applications). Let us consider yet another example.

"Sparse" CNFs. 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 $2^k/8k$ 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

$$d = \frac{2^k}{8k} \cdot k = \frac{2^k}{8}$$

other bad events. As

$$2^{-k} \cdot (\frac{2^k}{8} + 1) \leqslant \frac{1}{e},$$

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. $\square$

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 almost 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, run the subroutine `Fix`(C_i).

Subroutine `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 `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 `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 `Fix`(C_i) is also satisfied now. This means that *if* the call to `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

We are *not* going to case the analysis in terms of Entropy (given we have not covered necessary background on information theory) and thus we will attempt to do the analysis directly. For that, we just need the following information-theoretic lemma.

Lemma 4. *Let $f : \{0,1\}^m \rightarrow \{0,1\}^*$ be any fixed* injective *function (range of f can be arbitrarily large). Fix any $\delta \in (0,1)$. Suppose we sample x uniformly at random from $\{0,1\}^m$; then, with probability at least $1-\delta$, $f(x)$ has length at least $m - \log(1/\delta)$.*

This lemma says that if we attempt to "compress" the strings in $\{0,1\}^m$ to different lengths while being able to recover them from the compression, most of the times we barely manage to get a compression better than a small additive term. Even more informally speaking, *we cannot hope to compress random strings.*

Proof. The total number of strings with length $< m - \log(1/\delta) := t$ is

$$\sum_{i=1}^{t-1} 2^i < 2^t = \delta \cdot 2^m.$$

On the other hand, the function f is injective. Thus, there can be at most $\delta \cdot 2^m$ strings such that x is mapped to a string of length $< t$ by f. As such, for a randomly chosen x, the probability that x belongs to this set is at most δ, 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.*

Suppose Algorithm 1 calls the subroutine `Fix` s times in total. Then, how many random bits it 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 $n + s \cdot k$. 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 $\texttt{Fix}(C_i)$ 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 0.

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

  $$100, 2, 0, 5, 7, \cdots$$

  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 very final assignment output by the algorithm in n bits.

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.

So, what is the summary? The above compression scheme (which is a deterministic function of the random bits used by the algorithm), compresses

$$n + s \cdot k$$

random bits into a log of size

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

bits. This implies that with probability at least 0.99, we have,

$$m + s \cdot (\log R + O(1)) \geqslant s \cdot k - O(1).$$

But, the important thing is that $\log R + O(1)$ can still be $k - O(1)$ by the choice of R. Thus, we have,

$$m + s \cdot (k - O(1)) \geqslant s \cdot k - O(1)$$

which means

$$s = O(m).$$

In other words, with probability 0.99, the algorithm terminates only after $O(m)$ calls to `Fix`. This concludes the proof of Theorem 5.

References

[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. 2, 3