Chapter 24

Algorithmic Lovász Local Lemma and Entropy Compression

We study further aspects of the Lovász Local Lemma (LLL) in this chapter.

24.1 Proof of Lovász Local Lemma

In the previous chapter, 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[1] (notice that we effectively replace $1/e$ in the original statement with a weaker constant of $1/4$).

Theorem 24.1 (**"Weaker" LLL**). *Suppose $B_1, \ldots, B_n$ are 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* $T \subseteq [n] \setminus (N(i) \cup \{i\})$,

$$\Pr\left(B_i \mid \{B_j\}_{j \in T}\right) = \Pr(B_i);$$

(as in Theorem 23.3, conditioning on $\{B_j\}_{j \in T}$ *means any of the* $2^{|T|}$ *possible combinations of events* B_j *in* $j \in T$ *happening or not happening).*

Then, as long as

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

the probability that ***none of*** $B_1, \ldots, B_n$ *happens is strictly greater 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

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

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:

[1] The proof of the stronger form also follows exactly the same strategy, just with more detailed calculations.

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,

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

The only "real" inequality above is in dropping $\wedge A_{S \cap N(i)}$ (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, $A_{S \cap N(i)}$ contains at most 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 now 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 $N(i) \cap S$ has at least one element (otherwise the sum above is empty), 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 $p \cdot d \leqslant 1/4$.

Plugging the bounds of Step 2 into the last line of Step 1 gives us

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

as desired. This concludes the proof. □

It is worth mentioning that the statement of Theorem 24.1 is actually the original version of LLL proven by Erdős and Lovász in [EL75].

Another application of LLL: "Sparse" CNFs. Let us use Theorem 24.1 to showcase yet another application of LLL in preparation for the main topic of this chapter.

Recall that for any integer $k \geqslant 1$, a k-CNF (conjunctive normal form) is a set of m *clauses* each consisting of an 'OR' of k *literals* over n variables; moreover, these clauses are joined by '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 24.2. *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 a simple application of LLL. Suppose we pick the assignment to $x_1, \ldots, x_n$ uniformly at random 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 an 'OR' of k literals, there is only one assignment to its k variables that does not satisfy it. Thus,

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

On the other hand, each bad event B_i is entirely independent of all bad events B_j whose clauses do not share any variable with clause i. Since each variable in clause i can belong to at most $2^k/(4k)$ other clauses and there are k variables in 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

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

we can apply LLL in Theorem 24.1 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. □

24.2 An *Algorithmic* LLL?

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

For instance, in Proposition 24.2, 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 $\approx \left(1 - 2^{-k}\right)^n \approx 2^{-\Theta_k(n)}$. That requires running the algorithm some $2^{\Theta_k(n)}$ 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 (whose existence LLL guarantees). 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 the last two decades. We will see an application of this to finding a satisfying assignment in Proposition 24.2 in polynomial time due to a brilliant idea of [Mos09].

24.2.1 An Algorithmic Version of Proposition 24.2

The algorithm is as follows.

Algorithm 24.3.

1. Let $C_1, \dots, C_m$ be the clauses and x be a random assignment in $\{0,1\}^n$ to 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 k variables of clause C_i from $\{0,1\}^k$ uniformly at random.
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 $\texttt{Fix}(C)$.

Note that it is not clear a priori that this algorithm ever terminates. This is because we may be "fixing" one clause, but then create many violated clauses 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 it does find a satisfiable assignment.

Lemma 24.4. *Whenever Algorithm 24.3 terminates, the final assignment x to the variables satisfies all clauses.*

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

Now consider some iteration $i \geqslant 1$. By induction hypothesis, $C_1, \dots, C_{i-1}$ 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 $C_1, \dots, C_{i-1}$ 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 matters possibly even worse! This is where the brilliant idea of [Mos09] comes into the picture, which provides a *unique* way of analyzing these types of processes, called the **entropy compression** method.

24.2.2 Entropy Compression Method and Runtime Analysis of Algorithm 24.3

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

Lemma 24.5. *Let $\delta \in (0,1)$ and $f : \{0,1\}^a \to \{0,1\}^b$ be any fixed function with the following property: if we sample x uniformly at random from $\{0,1\}^a$, we can recover x from $f(x)$ with probability at least δ. Then, we should have $b \geqslant a - \log(1/\delta)$.*

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

Proof. Let $X \subseteq \{0,1\}^a$ be the set of strings that can be recovered uniquely from $f(x)$. The function f from $X \to \{0,1\}^b$ must be injective as otherwise we cannot recover x uniquely from $f(x)$ if $f(y) = f(x)$ also and both $x, y \in X$. This means that $|X| \leqslant 2^b$. 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,

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

concluding the proof. □

We are going to prove that Algorithm 24.3 is "trying" to compress random bits—thus, if, with large probability, it gets to run for a very long time, then it will manage to compress random bits beyond what is allowed by Lemma 24.5, a contradiction. In particular, we will prove the following main theorem.

Theorem 24.6 ([Mos09])**.** *There is a constant c such that the following holds for every integer k with $2^{k-c} \geqslant k$: if Φ is a k-CNF wherein every variable appears in at most $2^{k-c}/k$ other clauses, then Algorithm 24.3 finds a satisfying assignment of Φ in $O(m)$ calls to* `Fix` *with probability at least* 0.99*.*

Let s be a parameter to be determined later (we will 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.

First, how many random bits does the modified algorithm generate throughout the whole process? Well, it starts with n random bits 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 of 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.

- First, we will write a string of m bits where its i-th bit is 1 if in the i-th iteration of the for-loop of Algorithm 24.3, 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 the "top-level" calls to `Fix` were (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`(C_{100}) and then calls `Fix` on its second neighbor, and terminates, and then call `Fix` on its fifth neighbor and inside it call `Fix` on the seventh neighbor of this clause 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 calls, we simply pad the log with dummy strings to a fixed length as if it was not terminated).

We claim that using this information, we can recover *all* the random bits used by the algorithm exactly. This is because the n-bit suffix gives x after the last call; we can then 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 that 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 know what the values of x were at every step, and, since we also know the clauses, exactly which random bits were used.

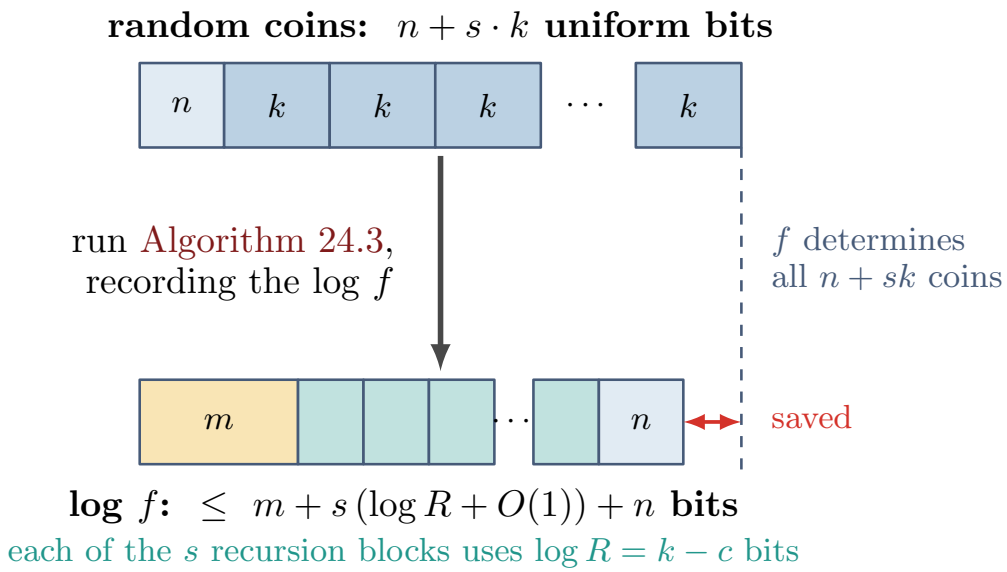

Figure 24.1: An illustration of entropy compression in Theorem 24.6.

Now, suppose that the original algorithm makes all s calls to `Fix` 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 24.5 for $a = n+s\cdot k$ (the coin tosses) and $b =$ the length of the long, we have that the length of the log should be at least

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

On the other hand, the length of the log is 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, 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 the hidden. constant in 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 within $m + \log(1/\delta)$ calls to `Fix` with probability at least $1 - \delta$, concluding the proof of Theorem 24.6.

Exercises

Exercise 24.1. The analysis of Theorem 24.6 shows that, for every $\delta \in (0,1)$, with probability at least $1 - \delta$ the algorithm makes at most $m + \log(1/\delta)$ calls to `Fix`. Let S denote the total (random) number of calls to `Fix` that Algorithm 24.3 makes on a k-CNF satisfying the hypothesis of Theorem 24.6.

(a) Show that $\Pr(S > m + t) \leqslant 2^{-t}$ for every $t \geqslant 0$, and use this to prove that $\mathbb{E}[S] \leqslant m + 2$.

(b) Assuming that each call to `Fix` does $O(k)$ work resampling and checking its own clause, and that the initial for-loop of Algorithm 24.3 costs $O(m \cdot k)$ time, prove that the *expected* running time of Algorithm 24.3 is $O(m \cdot k)$ time.

Exercise 24.2. We now give an alternative proof of the tail bound in Theorem 24.6 using a direct counting/union-bound argument in place of Lemma 24.5. As in the chapter, work with the modified algorithm that is halted after exactly s calls to `Fix`, and recall that its log consists of an m-bit prefix (the top-level indicators), a middle part that names the recursion using $\log R + O(1)$ bits per call, and an n-bit suffix (the final assignment); recall also that every random string leading to a completed length-s log can be reconstructed from that log.

(a) Show that the number of distinct logs the modified algorithm can output with exactly s calls to `Fix` is at most $2^{m + s(\log R + O(1)) + n}$.

(b) The modified algorithm is driven by $n + s \cdot k$ uniform coins. Using part (a) together with the recoverability of the coins, show that the probability that Algorithm 24.3 makes at least s calls to `Fix` is at most

$$\frac{2^{m + s(\log R + O(1)) + n}}{2^{n + s \cdot k}} = 2^{m - s(c - O(1))},$$

where we used $\log R = k - c$.

(c) Deduce that for a large enough constant c and $s = m + t$ this probability is at most 2^{-t}, recovering the conclusion of Theorem 24.6 without invoking Lemma 24.5.

Exercise 24.3 (♦). Let $H = (V, E)$ be a k-uniform hypergraph with $|V| = n$ vertices and $|E| = m$ edges, each edge being a set of k vertices. A 2-coloring $\chi : V \to \{0,1\}$ is *proper* if no edge is *monochromatic* (all k of its vertices receiving the same color). Suppose every edge shares a vertex with at most 2^{k-c} other edges, for some large constant $c > 10$. Design and analyze a resampling algorithm, in the spirit of Algorithm 24.3, that finds a proper 2-coloring of H.

(a) Under a uniformly random coloring $\chi \in \{0,1\}^n$, show that a fixed edge e is monochromatic with probability exactly 2^{1-k}. Then state the analogue of Algorithm 24.3: start from a uniform coloring and, while some edge is monochromatic, pick one (in a fixed order) and `Fix` it by resampling the colors of its k vertices, recursing on any neighbor edge that has become monochromatic.

(b) Adapt the compression scheme of the chapter to this setting, and show carefully that the log still recovers all of the coins. Explain why each `Fix` call now needs *one extra bit* beyond the $\log R + O(1)$ bits used in the CNF case. (Hint: unlike a violated clause, a monochromatic edge does not have a *unique* falsifying coloring of its k vertices—how many does it have?)

(c) Conclude that the algorithm finds a proper 2-coloring in an $O(m)$ expected resamplings, hence in $O(m \cdot k)$ expected time. Identify precisely where the argument uses that c is a large constant, and how large c must be for the per-call bit savings to be positive.