CS167: Reading in Algorithms
The Algorithmic Lovász Local Lemma[*]

Tim Roughgarden[†]

April 9, 2014

1 Positive-Probability Events

Let $\mathcal{E}_1, \mathcal{E}_2$ be two events — subsets of a common probability space. Suppose I tell you that $\mathbf{Pr}[\mathcal{E}_1], \mathbf{Pr}[\mathcal{E}_2] > 0$. Does this imply that $\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] > 0$? That is, if either A or B could happen separately, is it the case that A and B could both happen?

Not necessarily. E.g., with a single coin flip, if $\mathcal{E}_1$ represents "heads" and $\mathcal{E}_2$ "tails," then $\mathbf{Pr}[\mathcal{E}_1] = \mathbf{Pr}[\mathcal{E}_2] = \frac{1}{2}$ while $\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = 0$.

If $\mathcal{E}_1$ and $\mathcal{E}_2$ are independent, then sure, no problem:

$$\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \underbrace{\mathbf{Pr}[\mathcal{E}_1]}_{>0} \cdot \underbrace{\mathbf{Pr}[\mathcal{E}_2]}_{>0} > 0.$$

If $\mathbf{Pr}[\mathcal{E}_1], \mathbf{Pr}[\mathcal{E}_2] > \frac{1}{2}$, then the events must overlap and so $\mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] > 0$. Formally, the Union Bound implies that

$$\mathbf{Pr}[\neg\mathcal{E}_1 \cup \neg\mathcal{E}_2] \le \underbrace{\mathbf{Pr}[\neg\mathcal{E}_1]}_{<1/2} + \underbrace{\mathbf{Pr}[\neg\mathcal{E}_2]}_{<1/2} < 1, \tag{1}$$

so the complementary event $\mathcal{E}_1 \cap \mathcal{E}_2$ has positive probability.

The Lovász Local Lemma (LLL) states that as long as the events $\mathcal{E}_1, \ldots, \mathcal{E}_m$ are "not too dependent" and that each individual event has reasonably large probability, there is a positive probability that all m of the events occur simultaneously.

2 k-SAT

We focus on a canonical and illustrative special case of the LLL, the k-SAT problem.[1] Recall that a clause of a k-SAT formula is the disjunction of k literals, each of which is a variable or

[*]©2014, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 462 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]There are tons of other interesting applications of the LLL, see [1, 3].

its negation. For example, $x_3 \vee \neg x_7 \vee x_{10} \vee \neg x_{14}$ is a 4-SAT clause. We assume for convenience that every clause involves k distinct variables. You've probably thought primarily about 2-SAT and 3-SAT in the past; for this lecture, it's good to think of k as being a larger constant, say 8. We use m to denote the number of clauses and n the number of variables.

A few warm-up observations. Consider a single clause. It has k distinct variables. There are 2^k different truth assignments to these k variables. Observe that *only one* of them fails to satisfy the clause — every variable has to be assigned incorrectly. (E.g., in the example above, $x_3 = F$, $x_7 = T$, $x_{10} = F$, $x_{14} = T$.) The other $2^k - 1$ truth assignments satisfy the clause. You can think of a k-SAT clause as forbidding one particular assignment to a k-tuple of variables.

Fix a clause C. If we pick a truth assignment uniformly at random, then the probability that C is not satisfied is $1/2^k$, so the probability that it is satisfied is exactly $1 - 2^{-k}$. This trivial observation has some interesting corollaries. First, by linearity of expectation, the expected number of clauses satisfied by a random truth assignment is exactly $(1 - 2^{-k})m$. In particular, there exists a truth assignment that satisfies at least a $1 - 2^{-k}$ fraction of the clauses. (Prove both these statements.) This is a *lot* of clauses for even modest values of k. Satisfying lots of clauses of a k-SAT formula is trivial; satisfying all (or almost all) of them is what's hard. Second, suppose the number of clauses m is less than 2^k. Then, the Union Bound argument of (1) proves that all clauses are satisfied with probability at least $1 - m2^{-k} > 0$. Thus, a k-SAT formula with less than 2^k clauses is guaranteed to be satisfiable.

3 Statement of the LLL for k-SAT

The LLL allows us to conclude satisfiability for k-SAT formulas for an arbitrarily large number m of clauses. There must be a catch, since even 2^k clauses is enough to force unsatisfiability — just pick your favorite k variables and include all possible clauses for them, ruling out the 2^k possible truth assignments to them one-by-one. The idea is to let the number of clauses grow arbitrarily large while keeping the "amount of dependence" between the clauses bounded in the ballpark of 2^k.

Precisely, obtain a graph $G = (V, E)$ from a k-SAT formula as follows. The vertices V are the clauses. There is an edge $(i, j) \in E$ if and only if clauses i and j share a variable. If one clause contains x_i and another $\neg x_i$, this still counts as sharing a variable. Thus, the degree of a vertex corresponds to the number of overlapping clauses. Let $d = 1 + \max_{v \in G} \deg(v)$; the "+1" is because it's convenient to think of a clause as overlapping with itself.

For the rest of this lecture, we'll say that a k-SAT formula φ meets the *LLL condition* if the parameter d is at most $2^{k-3} = 2^k/8$. We'll prove the following.

Theorem 3.1 *If φ meets the LLL condition, then φ is satisfiable. Moreover, a satisfying assignment can be found in randomized polynomial time.*

The existential statement is from a paper of Erdös and Lovász [2]. There has been a long sequence of works that develop algorithmic versions (see [6]). Our goal here is to cover the

1. Let **x** denote a random assignment, with each of the n variables set independently to true or false (with 50/50 probability).

   [We will modify the global variable **x** as the algorithm proceeds.]

2. While some clause C of φ is violated by **x**, Fix(C).

Figure 1: The algorithm Alg. The input is a k-SAT formula φ that meets the LLL condition.

1. Randomly reassign the k variables of φ contained in C to true or false (independently, with 50/50 probability).

2. While some clause D of φ that shares a variable with C is violated by **x**, Fix(D).

   [Possibly $D = C$, if we unluckily re-chose the variables assignments that violate C.]

Figure 2: The subroutine Fix(C). The input is a clause C of φ that is violated by the current assignment **x**.

coolest proof of the lot, which is also one of the most recent, due to Moser and Tardos [4, 5].

4 The Algorithm

We analyze the randomized algorithm Alg in Figure 1, the workhorse of which is the subroutine Fix (Figure 2).[2]

Observe that Alg uses n random bits in the first step, to choose a random initial assignment, and k fresh random bits every time the subroutine Fix is called.

We can visualize the execution of Alg via a recursion tree, as in Figure 3. It will be useful to distinguish between the calls to Fix made by Alg— the first level of the recursion tree — and recursive calls to Fix in the lower levels of the recursion tree.

As is clear from the stopping condition, the responsibility of an invocation to Fix(C) is to modify the assignment **x** until the clause C and all the clauses overlapping with C are satisfied.

Lemma 4.1 *If the subroutine* Fix*(C) terminates, then it terminates with a variable assignment in which C and all clauses that share a variable with C are satisfied.*

The next lemma identifies a measure of progress made by terminating calls to Fix.

[2]It is audacious to propose such a simple algorithm for the legendary SAT problem. Perhaps it's no coincidence that Moser [4] was a graduate student when he proved the following results; a more seasoned researcher might have dismissed the approach out of hand.

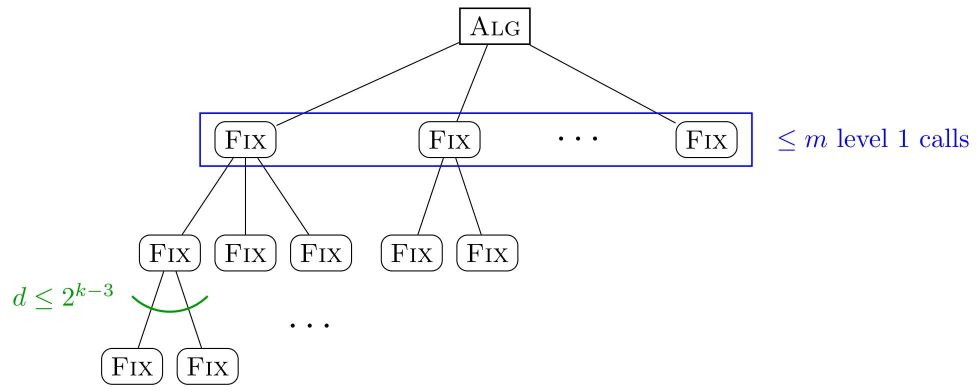

Figure 3: Recursion tree representing the algorithm's execution. ALG makes at most m calls to FIX, and each call to FIX recursively makes at most 2^{k-3} additional calls to FIX. The algorithm traverses the tree in depth first order.

Lemma 4.2 *A call to* FIX*(C) that terminates does not change any clauses of φ from satisfied to violated.*

Proof: Suppose clause C^* is satisfied by the current variable assignment when FIX(C) is called. If C^* shares a variable with C, then Lemma 4.1 implies that FIX(C) can only terminate with a truth assignment that satisfies C^*. If C^* shares no variables with C, then randomly reassigning C's variables cannot make C^* violated. Inductively, calls to FIX(D) by FIX(C) will also not make C^* violated.[3] In any case, C^* is again satisfied by the resulting truth assignment if FIX(C) terminates. ■

We can now prove correctness of ALG, conditioned on termination.

Lemma 4.3 *If the algorithm* ALG *terminates, then it terminates with a variable assignment in which every clause of φ is satisfied.*

Proof: Every call to FIX(C) by ALG— the level-1 nodes of the recursion tree in Figure 3 — fixes a violated clause of φ without creating any new violated clauses (Lemma 4.2). Thus these outer calls to FIX fix the clauses violated by the initial random assignment one by one, and if all the calls terminate then the result is a satisfying assignment. ■

The proof of Lemma 4.3 is essentially a monotonicity argument — the number of satisfied clauses is strictly increasing with the number of outer calls to FIX(C) by ALG. But note that the number of satisfied clauses is *not* strictly increasing throughout the algorithm — when the variables of a violated clause C are randomly reassigned, many other clauses might become newly violated.

In light of the above non-monotonicity, why should any given call to FIX ever terminate? Indeed, Lemma 4.3 implies that if we feed an unsatisfiable k-SAT formula into the algorithm

[3]The actual truth assignments to the variables of C^* might be different before and after the call to FIX(D), however — do you see why?

Alg, it will run forever. There's no reason to think that it won't run forever also on satisfiable instances. The surprising result is that it terminates with high probability — quickly, even — on k-SAT formulas that meet the LLL condition. This implies that every such formula is satisfiable.

Theorem 4.4 *For every k-SAT formula that meets the LLL condition, the algorithm* Alg *terminates (with a satisfying assignment) in polynomial time with high probability.*

5 Proof of Theorem 4.4

Here is a trivial observation. If a function $f : A \to B$ is injective — i.e., if it is invertible on its range $f(A)$ — then $|B| \geq |A|$. Our proof of Theorem 4.4 follows from this counting principle. Your instructor is not aware of any other non-trivial running time analyses that boil down to this principle.

Let φ be a k-SAT formula with m clauses that meets the LLL condition. We also assume that $m \geq 2^k$, since otherwise the formula is trivially satisfiable (see Section 2). For a parameter T, suppose that the probability that Alg finds a satisfying assignment within T calls to the subroutine Fix is zero. We prove that this is possible only if T is small; hence, if T is not too small, there is a positive probability that Alg finds a satisfying assignment within T calls to Fix, and in particular φ is satisfiable.

We now describe the sets A and B and the invertible function f. Recall that Alg uses n random bits to choose an initial truth assignment, and k fresh random bits every time Fix(C) is called, to randomly reassign the k variables in the clause C. If we abort Alg after T calls to Fix, then Alg uses a total of $n + kT$ random bits. We take the set A to be all possible choices of these random bits — the 2^{n+kT} bit-strings of length $n + kT$.

Next we describe the function f, using an algorithm — the output of f is the output of this algorithm. Looking ahead, invertibility of f will correspond to the computation of this algorithm being "reversible." The input to f is a triple $(\mathbf{x}_0, \mathbf{y}_0, \epsilon)$, where $\mathbf{x}_0$ is the initial variable assignment (n bits), $\mathbf{y}_0$ is a reservoir of kT bits to be used during the first T calls to Fix, and ϵ denotes the empty string. Note that after all the random bits $\mathbf{x}_0, \mathbf{y}_0$ needed by Alg have been fixed up front, Alg is a deterministic algorithm. The output of f is a triple $(\mathbf{x}_T, \epsilon, \mathbf{z}_T)$, where $\mathbf{x}_T$ is the current truth assignment of Alg after T calls to Fix, ϵ is the empty string, and $\mathbf{z}_T$ is a transcript of Alg's execution, detailed below.

The primary responsibility of the transcript is to keep track of the recursion tree (Figure 3) induced by the algorithm's execution — which Fix call invoked which other Fix call, and for which clause. Each call to Fix consumes k new random bits and appends some annotating bits to the transcript. It's easy to see how this works with an outer call to Fix(C), by Alg. If the current triple is $(\mathbf{x}, \mathbf{y}, \mathbf{z})$, we modify it to $(\mathbf{x}', \mathbf{y}', \mathbf{z}')$ as follows. Since this call to Fix(C) uses k fresh random bits to reassign the k variables in the clause C, $\mathbf{y}'$ is just $\mathbf{y}$ with the leading k bits removed. Similarly, $\mathbf{x}'$ is obtained from $\mathbf{x}$ by overwriting the k variables of C with the values of these random bits. Finally, we append to the transcript $\mathbf{z}$ a total of $\lceil \log_2 m \rceil + 1$ bits; the first bit is a "1" (indicating that we're pushing a new Fix record on the program stack) and the remaining $\lceil \log_2 m \rceil$ bits are a binary encoding of the index of the clause C.

We'll be more clever in logging recursive calls to FIX — indeed, this is the crucial point where the LLL condition is used. Suppose FIX(D) is called recursively by FIX(C). By the definition of FIX, this only happens when C and D share a variable. The key idea is: rather than recording D's index in the list of all clauses, we only record D's index *within the subset of clauses that share a variable with* C. Formally, we again obtain $\mathbf{y}'$ from $\mathbf{y}$ by removing the leading k bits, $\mathbf{x}'$ from $\mathbf{x}$ by overwriting the values of the k variables in D with the values of these k bits, and we append $k-2 < \lceil \log_2 m \rceil$ bits to the transcript $\mathbf{z}$, a "1" followed by a binary encoding of D's index in the list of (at most 2^{k-3}, by the LLL condition) clauses that share a variable with C.

One extra detail completes the definition of f: when a FIX record is popped off of the program stack, a "0" is appended to the transcript $\mathbf{z}$ ($\mathbf{x}$ and $\mathbf{y}$ remain unchanged).

Here are the two important lemmas.

Lemma 5.1 *The function f is invertible. Equivalently, for every final truth assignment $\mathbf{x}_T$ and transcript $\mathbf{z}_T$, there is at most one choice for an initial assignment $\mathbf{x}_0$ and a reservoir $\mathbf{y}_0$ of kT bits such that $f(\mathbf{x}_0, \mathbf{y}_0, \epsilon) = (\mathbf{x}_T, \epsilon, \mathbf{z}_T)$.*

Lemma 5.2 *The length of the final transcript $\mathbf{z}_T$ is always at most $m(\lceil \log_2 m \rceil + 2) + T(k-1)$.*

Before proving these two lemmas, let's see why they imply Theorem 4.4.

Proof of Theorem 4.4: An input to the function f is n bits $\mathbf{x}_0$ and kT bits $\mathbf{y}_0$. Thus, there are exactly 2^{n+kT} inputs. Since f is invertible (Lemma 5.1) it has exactly 2^{n+kT} outputs. On the other hand, every output of f is described by the n bits $\mathbf{x}_T$ and the at most $m(\lceil \log_2 m \rceil + 2) + T(k-1)$ bits of $\mathbf{z}_T$ (Lemma 5.2). Hence,

$$2^{n+kT} \leq 2^{n+m(\lceil \log_2 m \rceil + 2) + T(k-1)};$$

taking logarithms (base 2) and rearranging, we obtain

$$T \leq m(\lceil \log_2 m \rceil + 2). \tag{2}$$

Summarizing, ALG fails to terminate on a formula satisfying the LLL condition within T calls to FIX with probability 1 only if (2) holds. Thus, there is a positive probability that ALG terminates within $m(\lceil \log_2 m \rceil + 2) + 1$ calls to FIX; in particular, φ is satisfiable.

The exact same argument shows that ALG fails to terminate on a formula satisfying the LLL condition within T calls to FIX with probability at least 2^{-c} only if $T \leq m(\lceil \log_2 m \rceil + 2) + c$. (You should check this.) Thus, ALG terminates (with a satisfying assignment) on such a formula within $O(m \log m)$ FIX calls, and hence in polynomial time, with high probability. ■

We now prove Lemmas 5.1 and 5.2.

Proof of Lemma 5.1: Fix a final truth assignment $\mathbf{x}_T$ and transcript $\mathbf{z}_T$. We give a deterministic algorithm that reconstructs the unique initial assignment $\mathbf{x}_0$ and reservoir $\mathbf{y}_0$ of kT bits that leads to this output of f (or determines that no such input exists). This algorithm will reverse the steps of the algorithm that defines f, one by one.

The first observation is that the recursion tree (Figure 3) of algorithm ALG is uniquely defined by the transcript $\mathbf{z}_T$, including the identify of the clause C passed to each call to FIX. This is essentially depth-first search, like tracing through a program step-by-step with a debugger. At the beginning of the transcript, ALG is at the root of the recursion tree. The first bit of the transcript should be a "1;" otherwise $\mathbf{z}_T$ is not a possible output of f. The subsequent $\lceil \log_2 m \rceil$ bits describe the identify of the clause C that corresponds to the first call to FIX. If the next bit is a "0," then this call to FIX makes no recursive calls, and control returns to the root node of the recursion tree. If the next bit is a "1," then this indicates a recursive call FIX(D) by FIX(C). The next $k-2$ bits encode the index of clause D in the list of clauses that share a variable with C. Given that we know C at this point of tracing through the transcript $\mathbf{z}_T$, we can uniquely identify the clause D. Continuing this process reconstructs the entire recursion tree corresponding to the first T FIX calls.

Given the recursion tree, we reconstruct $\mathbf{x}_0$ and $\mathbf{y}_0$ from $\mathbf{x}_T$ and $\mathbf{z}_T$ by undoing the FIX calls one by one. First, from the recursion tree, we can identify the clause C on which the Tth call to FIX was made. The key observation is: FIX *is only called on violated clauses, and a clause has a unique violating assignment.* For example, if the Tth FIX call was on the clause $x_3 \vee \neg x_7 \vee x_{10} \vee \neg x_{14}$, then we know that prior to the call the truth assignment $\mathbf{x}_{T-1}$ had $x_3 = F$, $x_7 = T$, $x_{10} = F$, and $x_{14} = T$. The upshot is we can reconstruct the state before the Tth call to FIX: we prepend the values of the variables of C in $\mathbf{x}_T$ to the (currently empty) reservoir $\mathbf{y}_T$, and reconstruct $\mathbf{x}_{T-1}$ from $\mathbf{x}_T$ by overwriting the variables of C with the unique truth assignments that cause C to be violated. Iterating back through all the FIX calls in this way, we recover the original truth assignment $\mathbf{x}_0$ and reservoir $\mathbf{y}_0$. ■

Proof of Lemma 5.2: Bits are added to the transcript $\mathbf{z}$ only when a FIX record is pushed onto or popped off of the program stack. Lemma 4.2 implies that every call to FIX(C) by ALG — a level-1 node of the recursion tree in Figure 3 — fixes a violated clause without creating any new violated clauses. Thus, there are at most m such calls to FIX. Each such call contributes $\lceil \log_2 m \rceil + 1$ bits to $\mathbf{z}$ when the corresponding record is pushed on the stack; if the call terminates before ALG is aborted, it contributes another bit when its record is popped off the stack. By definition, there are at most T recursive calls to FIX. Each of these contributes at most $(k-2)+1 = k-1$ bits to $\mathbf{z}$. Summing over all of these contributions to $\mathbf{z}$ proves the lemma. ■

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method.* Wiley, 2008. Third Edition.

[2] P. Erdös and L. Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In A. Hajnal, L. Lovász, and V. Sós, editors, *Infinite and Finite Sets*. North-Holland, 1975.

[3] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge, 2005.

[4] R. A. Moser. A constructive proof of the Lovász local lemma. In *Proceedings of the 41st ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2009.

[5] R. A. Moser and G. Tardos. A constructive proof of the general Lovász local lemma. *Journal of the ACM*, 57(2), 2010.

[6] A. Srinivasan. New constructive aspects of the Lovász Local Lemma, and their applications. Scribe notes from Princeton Approximation Algorithms Workshop, 2011.