5.3 Lecture 28 (7/Dec): Moser-Tardos branching process algorithm for the local lemma

Now we describe an algorithm for finding satisfying assignments to the local lemma. The algorithm works in great generality and achieves the same limiting threshold (whenever the algorithm is applicable) as the full local lemma; however, for simplicity, we describe it here in a slightly restricted setting. (Most notably the dependency graph will be symmetric.)

Let $H = (V, E)$ be a SAT instance (see definitions in Sec. 5.1); we can encode most applications of the local lemma in these terms. As before, say that two clauses are neighbors in the dependency graph if they share any variable (not necessarily literal). Write $n = |V|$ (number of variables), $m = |E|$ (number of clauses). Call the clauses $T_1, \ldots, T_m$ according to an arbitrary ordering.

We have from last time a corollary of the local lemma:

Corollary 98 *Suppose every clause in $T \in E$ has size k and has at most d neighbors. If $d + 1 \leq 2^k/e$ then H is satisfiable.*

Moser-Tardos Algorithm [78, 79]
Pick a random assignment to V
While there is an unsatisfied clause, pick the first-indexed such clause T and run Fix(T).

Fix(T)
Recolor the variables of T u.a.r. until it is satisfied.
While T has an unsatisfied neighbor, pick the first-indexed such neighbor T' and run Fix(T').

("First-indexed" is a mere convenience, any deterministic order is ok, even depending on the history of the algorithm so far). Observe that Fix implements a recursive or stack-based exploration analogous to Depth First Search (DFS), but it is possible for clauses to be revisited.

Theorem 99 *If $4(d + 2) \leq 2^k$ then the Moser-Tardos algorithm finds a satisfying assignment to H in time $\tilde{O}(n + mk)$.*

We are being loose in this presentation about the leading constant of 4 and about $d + 1$ rather than $d + 2$. These can be improved to e and $d + 1$.

We're also being a bit loose about the run-time. For a bound of $O(n + mk)$ we'll just keep track of the number of random bits the algorithm uses. The actual run-time, which includes data structure management, will be a little larger but only by some factor of about $\log nm$.

Before presenting the proof, let's see why what we are studying is very similar to a branching process. Fixing some clause as the root, there is an implicit tree extending out first to neighboring clauses, then to neighbors of those, and so on. (Of course there may be repetition but that works out in our favor.) The degree of this tree is $d + 1$, but our DFS needs to explore only a subtree of it, generated at random, in which the expected number of children of a node is bounded by $(d + 1)/2^k < 1$. So, intuitively, what is going on is that a Fix call that is initiated by the main procedure, tends to terminate after generating a finite DFS tree.

This is only of course intuition, and the formal proof follows.

Proof: The algorithm is implemented with the aid of an (infinite) random bit string $z = z_1 \ldots$. The first n bits are used for the initial assignment. Then, successive bits are used in batches of k for the Fix procedure.

The choice of z amounts to uniformly choosing a path down a non-degenerate binary tree (no vertices with one child), whose leaves represent successful terminations of the algorithm. (Note,

this is the tree of random bits, and we only descend in it—it is not the graph of clauses in which we are performing a DFS-like process!) Of course the tree is infinite (we might endlessly sample bits badly). However, we will argue that with high probability we reach a leaf fairly soon.

A key observation is that a call to Fix(T) has the following monotonicity property. If Fix(T) terminates, then after termination (a) T is satisfied, (b) any clause T' that was satisfied before the call is still satisfied.

(a) is obvious from the text of the procedure. For (b), consider the last time after Fix(T) started, at which any of the variables inside T' were changed. This change, which left T' unsatisfied, occurred during a call to Fix(T'') where T'' is T' or one of its neighbors. But as we see in the procedure, we cannot have terminated Fix(T'') while T' is unsatisfied. So (since this is all on a stack) we also cannot have terminated Fix(T).

Hence, the *main* procedure calls Fix at most m times on any z.

Let N_t be the number of nodes that the algorithm tree has at depth t. Since the algorithm always runs the first n steps and then operates in batches of k bits, leaves of the tree occur at the levels $n + sk$, after s calls to Fix (whether from the main procedure or recursively from within Fix).

For any such node which actually exists in the tree, what is the probability that the algorithm reaches it? Since all random seeds z are equally likely, this probability is precisely 2^{-n-sk}.

So what is the expected runtime of the tree? It is the sum of the probabilities that we reach each node. Namely,

$$\sum_{t \geq 0} N_t 2^{-t} = n + k \sum_{s \geq 1} 2^{-n-sk} N_{n+sk}.$$

You can see that if we had, say, $N_{n+sk} = 2^{n+sk}$, this sum would diverge, as of course it should, since that tree has no leaves at all. But even if there are some leaves, the sum can readily diverge. We need to show the tree is thin enough that the sum converges.

The method is to devise an alternative way of naming a vertex at level $t = n + sk$. The obvious way is to give the bits $z_1 \dots z_t$, but that allows us to name 2^t vertices, and so will not do. Our naming scheme must give all vertices distinct names, yet be such that the name space for vertices at depth t is considerably smaller than 2^t.

Call the vertex we are focusing on $Z = (z_1 \dots z_{n+sk})$. Suppose that rather than being told all the bits $z_1 \dots z_{n+sk}$, we're instead told, in order, the arguments (names of clauses) to Fix; plus the assignment to all the variables at the time we reach Z.

Then we can determine $z_1 \dots z_{n+sk}$ by "working backwards." The last clause, before its recoloring, had to have been in its unique unsatisfied assignment. In turn, the penultimate clause, before its recoloring, had to have been in its unique unsatisfied assignment. And so forth.

How many bits are required to specify Z in this alternative way?

1. n bits for the last assignment.
2. We list, in chronological order, each clause as it is pushed on the stack (i.e., is called in Fix) and when it is done (popped off the stack). Since subsequent Pushes are neighbors in the dependency graph, this requires only $\lg(d+2)$ bits per call (reserve one symbol for "Popping"). When we Pop all the way out to the main procedure (which is something we know has occurred from keeping track of the stack), we just need one bit per each of clause T_i, to indicate whether the main procedure calls Fix(T_i).

So,

$$\lg N_{n+sk} \leq \min\{n + sk, n + m + s\lceil \lg(d+2) \rceil\}.$$

Now, as above measuring runtime in terms of how many bits of z we read, we have,

$$\begin{aligned} E(\text{runtime}) &= n + k\sum_{s\geq 1} 2^{-n-sk} N_{n+sk} \\ &\leq n + k\sum_{s\geq 1} 2^{-n-sk} 2^{n+\min\{sk, m+s(1+\lg(d+2))\}} \\ &= n + k\sum_{s\geq 1} 2^{\min\{0, m+s(1+\lg(d+2)-k)\}} \end{aligned}$$

Since we have assumed $k \geq 2 + \lg(d+2)$, this is

$$\begin{aligned} &\leq n + k\sum_{s\geq 1} 2^{\min\{0, m-s\}} \\ &= n + k\sum_{s=1}^{m} 1 + k\sum_{s>m} 2^{m-s} \\ &= n + k(m+1). \end{aligned}$$

□