1 A simple algorithm for the Lovasz Local Lemma

Previously, we discussed the Lovasz Local Lemma (LLL), which (roughly) states that there might exist a situation where not a single bad event happens; here, we discuss a (simple) algorithm for actually computing a configuration of those events (i.e., an assignment to variables that those events are defined on) such that all events are satisfied.

Here is the setup. Suppose we have $m$ events $\{A_1, \ldots, A_m\}$ that are functions of the $n$ (deterministic) binary variables $X_1, \ldots, X_n$, $X_i \in \{0,1\}$, $i = 1, \ldots, n$. To express the relationship between the events and variables we write that event $A_i$ depends on vbl $A_i$, $i = 1, \ldots, m$; so, vbl $A_i$ might equal, say, $\{X_2, X_4\}$.

Now, further suppose that the $A_i$s satisfy the LLL hypothesis (i.e., Eqs. 1 or 3 in the notes for Lecture 14) and that we have a dependency graph, where $A_i$ is connected to $A_j \iff A_i$ and $A_j$ depend on (some of) the same variables, i.e., $\text{vbl} A_i \cap \text{vbl} A_j \neq \emptyset$.

One simple algorithm for computing an assignment to the $X_i$ is then given in Alg. 1.

\begin{algorithm}
\textbf{Input:} a set of events $\{A_1, \ldots, A_m\}$
\textbf{Initialization:} set $X_1, \ldots, X_n$ randomly;
\textbf{while} there exists a violated event \textbf{do}
  \hspace{1em} draw a violated event $A$ uniformly at random;
  \hspace{1em} draw a variable $X$ uniformly at random from vbl $A$;
  \hspace{1em} flip $X$’s value
\textbf{end}
\textbf{Output:} assignments for $X_1, \ldots, X_n$, which satisfy $A_1, \ldots, A_m$

\textbf{Algorithm 1:} A simple algorithm for the LLL.

Observe that Alg. 1 must return a right answer, since if/when it terminates it must return a satisfying assignment of the $X_i$; so, the question is: does this algorithm terminate? Flipping $X$ in order to fix $A_i$ might break $A_j$ if $A_i$ and $A_j$ both depend on $X$, so the answer may not be clear. We will need a new data structure to answer these questions, witness trees, which we discuss in the next section. (It is also interesting to note that the “flip $X$’s value” line in the algorithm here is kind of like a local search.)

15-1
2 Witness trees

A witness tree is a graph that attempts to summarize the order in which we drew (and fixed) violated events in Alg. 1. Here is how they work. Suppose we have the dependency graph given in Fig. 1, top and that we drew the violated events in (i.e., the first line in the while loop in Alg. 1 returned) the following order:

\[ C, A, E, C, E, B, A, D. \]  

(1)

Now, suppose we want to construct a witness tree for event \( D \). We proceed by working backwards:

1. First, we find the last occurrence of \( D \) (i.e., start at the end of list (1) and work to the left). Now, since we fixed \( D \), we draw a vertex for \( D \) in the witness tree (see Fig. 1, bottom).

2. Now, we ask: “could flipping \( A \) have broken \( D \)?” The answer is no (not directly), since \( A, D \) aren’t neighbors in the dependency graph; so, we don’t do anything, and keep moving to the left.

3. Next, we ask: “could flipping \( B \) have broken \( D \)?” (We don’t ask “could flipping \( B \) have broken \( A \), since \( A \) is not in the witness tree (yet).) The answer is yes, since \( B, D \) are neighbors in the dependency graph; so, the rule we follow is to make \( B \) a child of \( D \) in the witness tree.

4. We continue: “could flipping \( E \) have broken \( D \)?” No.

5. “Could flipping \( C \) have broken \( D \)?” Yes; we make \( C \) a child of \( D \).

6. “Could flipping \( E \) have broken \( D \)?” Yes, actually, because flipping \( E \) could have broken \( C \), which could have broken \( E \), which could have broken \( B \), which could have broken \( D \); so, the other rule we follow is if the current event in question is a neighbor in the dependency graph of any variable in the (current iteration of the) witness tree, then we attach the current variable in question to (i.e., make it a child of) that event in the witness tree, and we attach it as deeply as possible.

7. “Could flipping \( A \) have broken \( D \)?” Yes, because we could have \( A \implies B \implies D \); so, we make \( A \) a child of \( B \), where we follow the rule that if an event that we are going to add to the witness tree is connected to multiple events in the dependency graph, and those events are at equal depth in the witness tree, then we break ties arbitrarily in terms of where we add the current event.

8. “Could flipping \( C \) have broken \( D \)?” Yes; so, we make \( C \) a child of \( A \).
Figure 1: Top: a dependency graph for a set of events. Bottom: a witness tree for event $D$ based on the sampling order given in Eq. 1.
Variable | Initial values | Values after 1st flip | 2nd flip | ...  
--- | --- | --- | --- | ---  
$X_1$ | 1 | 0 | 0 | ...  
$X_2$ | 1 | 1 | 0 | ...  
$X_3$ | 1 | 1 | 1 | ...  
... | ... | ... | ... | ...  
$X_n$ | 1 | 0 | 0 | ...  

Table 1: A randomness table.

3 The randomness table

Next, we introduce the randomness table, another data structure that helps with the analysis of Alg. 1 using witness trees. Observe that if we knew ahead of time the variable(s) on each iteration of the algorithm that we would draw and fix, then we could construct Table 1, where each row is one of the $X_i$, and each column contains the values for (all of) the $X_i$ on an iteration of the algorithm ($X_i$ that did not have their values flipped carry the same values across columns); so, e.g., the first column of Table 1 contains the initial values of the $X_i$, the second column indicates that we flipped the values of $X_1, X_n$, the third column indicates that we flipped the values of $X_1, X_2, \ldots$.

Definition 1 (consistency) We say that a witness tree $\tau$ is consistent with a randomness table $T$ if plugging the values in $T$ into $\tau$ bottom-up, i.e., taking the values in the first column of $T$ and plugging them into (the variables that define) the deepest node in $\tau$, then plugging the values in the second column of $T$ into the second-deepest node in $\tau$, \ldots (up until the last column of $T$), causes all the nodes/events in $\tau$ to be violated (if these values caused the nodes/events to be satisfied on the other hand, then we wouldn’t have flipped the values).

Observe that any witness tree $\tau$ that is produced by first generating a randomness table $T$ ahead of time, and then populating its values bottom-up is — by definition — consistent with $T$.

Via the randomness table construct, we can say that the probability of generating a very large witness tree is very small; we will use this in our subsequent analysis.

Lemma 2 (concentration) Let $\tau, T$ be a witness tree and randomness table, respectively. Also, for simplicity, let each event be defined on $k$ variables (e.g., we can think of each event being a clause). Then

$$\operatorname{Prob}(\tau \text{ is consistent with } T) = 1/2^{k|\tau|},$$

where $|\tau|$ is the size of $\tau$.

Proof By definition, the event $\tau$ is consistent with $T$ is equivalent to the event $T$ violates the nodes in $\tau$. The total number of configurations of a node
in $τ$ is $2^k$; thus, the probability that a single node will be violated is $1/2^k$, and the probability that all nodes are violated is $(1/2^k)^{|π|}$, as claimed. ■

If we produce a witness tree from the randomness table on each iteration of the algorithm, then we can say that

$$E[\text{runtime of Alg. 1}] \leq E[\# \text{ of consistent witness trees}]$$

$$= \sum_{\text{w.t. roots } A} \sum_{\text{w.t. sizes } s=1}^{\infty} \text{Prob}(\text{w.t. rooted at } A \text{ and of size } s \text{ is consistent})$$

$$= \sum_{A} \sum_{s=1}^{\infty} 1/2^{ks}$$

$$\leq m \sum_{s=1}^{\infty} \left(\frac{1}{2}e^{(\Delta + 1)s}\right) \quad (\Delta \text{ is the maximum degree of our dependency graph})$$

$$\leq m \sum_{s=1}^{\infty} (a \text{ constant } < 1)^s \quad (\text{this is a convergent geometric series})$$

$$= O(m),$$

where we only sketched some of the details here (we ran out of time in class, and deferred the full proof until next time) — the main idea is that the runtime of Alg. 1 is $O(m)$. 