Lovász Local Lemma: A Survey of Constructive and Algorithmic Aspects, with an Application to Packet Routing

CPSC 536N - Term Porject

Bader N. Alahmad*

1 Introduction

The Lovász Local Lemma (LLL) is a mathematical statement that gives hope in a complex world of interdependent events. It says, briefly, that bad events can be avoided altogether with positive probability even if each is highly likely to occur, but will not certainly occur. There is even more to it: The bad events can be dependent, and with limited dependence, still they can be avoided.

In its original guise, the Lemma is non-constructive, in that it asserts that a collection of bad events can be avoided, or not, but in no way does it give a method for finding the points in the sample space which lie in the intersection of the complements of those bad events, if any. Non-constructive mathematical assertions of this sort are typical of the *probabilistic method* , where the existence of certain objects of interest is established as an event over a suitable probability space, with this event occurring with positive probability.

In this article, we will present and discuss an efficient randomized algorithm, designed by Moser and Tardos [5], for finding exactly those points in the sample space that do not belong to all the bad events simultaneously, if any, thus giving us a route for avoiding all such events.

Without further ado, let us get to the crux of the matter. Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ be a finite collection of events over a probability (measure) space, say $(\Omega, \mathcal{F}, \mathbb{P})$. Then

$$\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) = 1 - \mathbb{P}(A_1 \cup \cdots \cup A_m) \geqslant 1 - \sum_{i=1}^{m} \mathbb{P}(A_i) \tag{1}$$

by the finite additivity of probability measures. We start off by demanding a less ambitious requirement than the one we aim at presenting in this article: We are interested in showing that, under certain conditions, $\mathbb{P}(A_1^c \cap \cdots \cap A_m^c)$ is strictly positive. What the latter means is that, if the events A_i are "bad" events, then we can avoid the situation where the bad events take place simultaneously (i.e., achieve $\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) > 0$). This amounts to saying that we would like to make sure that the set $\{A_1^c \cap \cdots \cap A_m^c\}$ is not empty, and that $\mathbb{P}(\{\omega : \omega \in A_1^c \cap \cdots \cap A_m^c\}) > 0$. Our

*Ph.D. student, Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC, Canada; bader@ece.ubc.ca.

setup, however, consists of a finite sample space, so null sets (those of measure zero) are not a concern, and thus the condition that $\{A_1^c \cap \cdots \cap A_m^c\} \neq \emptyset$ suffices. Consider the simpler case when the events $A_1, \ldots, A_m$ are *symmetric* with respect to their likelihood of occurrence, i.e., $\mathbb{P}(A_i) \leqslant p$ for some $p \in [0, 1)$, for all $i \in \{1, \ldots, m\}$. Then the RHS of Eqn. (1) stipulates that $mp < 1$ is a necessary condition for $\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) > 0$ to hold. Therefore $p < 1/m$ is necessary. The last condition says that as the number of events increases, the probability of occurrence of each must become smaller (drastically in m) in order to achieve $\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) > 0$. The main reason our condition on the event probabilities is stringent is that our formulation so far does not consider independence; we are assuming nothing about how the events are dependent among each other. The situation becomes better when we know the maximum number of events upon which any event in $\mathcal{A}$ depends; call this number d. At another extreme, if the events are mutually independent, meaning that for every $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, m\}$, $\mathbb{P}\left(\bigcap_{j=1}^k A_{i_j}\right) = \prod_{j=1}^k \mathbb{P}(A_{i_j})$, then trivially

$$\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) = \prod_{i=1}^m (1 - \mathbb{P}(A_i)) = \prod_{i=1}^m (1 - p), \tag{2}$$

which is strictly positive if, and only if, $p < 1$ (we have assumed in doing so that if $A_1, \ldots, A_m$ are independent, then $A_1^c, \ldots, A_m^c$ are independent. This fact needs proof, which we provide in Appendix A.) If we know that every event depends on at most d other events in $\mathcal{A}$, then Erdős and Lovász [3] proved that $4pd \leqslant 1$ is sufficient for $\mathbb{P}(A_1^c \cap \cdots \cap A_m^c) > 0$ to hold. Spencer [8] weakened the condition above to

$$p\frac{(d+1)^{d+1}}{d^d} \leqslant 1.$$

Since $p\frac{(d+1)^{d+1}}{d^d} = p(d+1)\left(1 + \frac{1}{d}\right)^d \leqslant ep(d+1)$ (e is the base of the natural logarithm), it follows that if $ep(d+1) \leqslant 1$, then it must be the case that $p\frac{(d+1)^{d+1}}{d^d} \leqslant 1$. Thus the condition $ep(d+1) \leqslant 1$ is sufficient. Finally, Shearer [7] further weakened the condition to

$$p\frac{d^d}{(d-1)^{d-1}} \leqslant 1,$$

for $d \geqslant 2$ and $p \leqslant 1/2$ for $d = 1$, from which it follows that $epd \leqslant 1$ is sufficient.

To demonstrate the applicability of the symmetric LLL, consider as example the problem of coloring the vertices of a hypergraph with two colors, say red and blue. We say a hypergraph is 2-colorable if there exists a coloring of its vertices such that no edge has a single color for all of its vertices (we call an edge whose vertices are all colored by the same color *monochramatic*). Formally, given a hypergraph $G = (V, E)$, and assuming that every $e \in E$ has at least k vertices, is there a coloring $c : V \to \{r, b\}$ such that no $e \in E$ is monochromatic? It turns out that such a coloring exists if the following condition holds: every edge in E does not intersect with more than $2^{k-1}/e$ other edges (here e is the base of the natural logarithm). Consider the following simple randomized procedure for coloring the vertices V: set the color of every $v \in V$ independently and uniformly at random to one of $\{r, b\}$ (by flipping a coin for every vertex independently). Associate with every $e \in E$ the bad event A_e, where $A_e \stackrel{\text{def}}{=} e$ is monochromatic. Then we would like to avoid the situation where all edges are monochromatic, which translates to requiring that

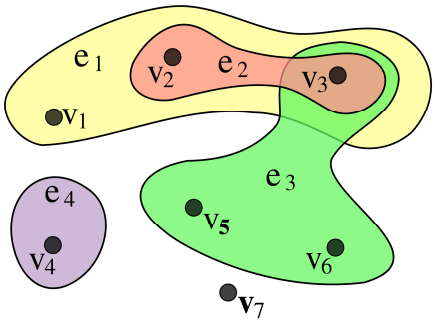


Figure 1: Example Hypergraph with 4 edges and 7 vertices. Edges e_1, e_2 and e_3 instersect at v_3. (http://en.wikipedia.org/wiki/Hypergraph).

$\mathbb{P}\left(\bigcap_{e\in E} A_e^c\right) > 0$. Let $V(e)$ denote the set of vertices that edge e contains. Since our algorithm sets the color of every vertex independently, it follows that

$$\begin{aligned}\mathbb{P}(A_e) &= \mathbb{P}(e \text{ is monochromatic}) \\ &= \mathbb{P}(c(v) = r \text{ for every } v \in V(e)) + \mathbb{P}(c(v) = b \text{ for every } v \in V(e)) \\ &\leqslant 1/2^k + 1/2^k = 1/2^{k-1},\end{aligned}$$

where the last inequality follows because every edge has at least k vertices. Now the condition that every edge intersects with at most $2^{k-1}/e$ other edges implies that an edge being monochromatic is affected by the coloring of at most $2^{k-1}/e$ other edges. Thus every event A_e is dependent on at most $2^{k-1}/e$ other events, and it follows that our dependency parameter is $d = 2^{k-1}/e$. Thus with $p = 1/2^{k-1}$, Shearer's condition $epd \leqslant 1$ is satisfied: $e\frac{1}{2^{k-1}}\frac{2^{k-1}}{e} \leqslant 1$, and the result follows.

In this article, however, we are concerned with an algorithm that is actually capable of "efficiently" finding the set of ωs that constitute $\{A_1^c \cap \cdots \cap A_m^c\}$. To prepare for a more general setting, we turn our attention to the asymmetric, general form of the LLL.

2 Algorithmic Lovász Local Lemma

2.1 Theoretical Setup

The difficulty of applying the LLL lies in pinpointing the dependencies between the bad events that we want to avoid. In order to facilitate "visualizing" the dependencies, we encode the bad events and their dependencies in the so-called *dependency graph*. Given a set $\mathcal{A} = \{A_1, \ldots, A_m\}$ of events in which every event depends on at most d other events, we construct a graph $G = (V, E)$ whose vertices are the events, i.e., $V = \mathcal{A}$ with $|V| = m$, and an edge exists between two vertices, say A_i and A_j, iff A_i is dependent upon A_j. Thus the degree of every vertex is at most d. For every $i \in \{1, \ldots, m\}$, let $N(A_i)$ denote the set of neighbors of A_i in G. Every event is dependent on itself, because $\mathbb{P}(A_i) = \mathbb{P}(A_i \cap A_i) = \mathbb{P}(A_i)\mathbb{P}(A_i)$ is satisfied either when $\mathbb{P}(A_i) = 0$, which does not happen in our setting (we excluded null events), or when $\mathbb{P}(A_i) = 1$, which we assumed does not occur in our setting as well. Define $N^+(A_i) = N(A_i) \cup \{A_i\}$. Then A_i is mutually independent of $\mathcal{A} \setminus N^+(A_i)$.

For example, consider the problem k-CNF. We are given as input a Boolean formula $\phi = c_1 \wedge \cdots \wedge c_m$ of m clauses over n variables $x_1, \ldots, x_n$, where each clause is the disjunction of exactly k literals. The problem is determine whether or not ϕ is satisfiable. It is known that k-CNF is polynomial-time decidable when $k = 2$, and is NP-Complete for $k \geqslant 3$. Let A_i be the event that clause c_i is NOT satisfiable, $i \in \{1, \ldots, m\}$. So A_i is a bad event that we need to avoid, for all i, so that ϕ is satisfiable, and thus $\{A_1^c \ldots, A_m^c\} \stackrel{\text{def}}{=} \phi$ is satisfiable. At the worst, every variable in every clause will appear in all the other $m-1$ clauses, and thus every assignment of every variable in clause c_i will affect the value of all the other $m-1$ clauses, thus $d \leqslant k(m-1)$. For instance, the following 3-CNF formula

$$\phi = \underbrace{(x_1 \vee x_2 \vee x_3)}_{c_1} \wedge \underbrace{(x_1 \vee \overline{x}_3 \vee x_4)}_{c_2} \wedge \underbrace{(x_4 \vee x_5 \vee \overline{x}_6)}_{c_3} \wedge \underbrace{(x_7 \vee \overline{x}_8 \vee x_9)}_{c_4} \tag{3}$$

has the dependency graph shown in Figure 2.

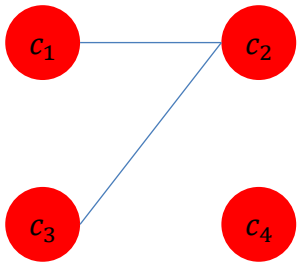


Figure 2: Dependency graph of ϕ in Eqn. (3).

Next we state the general form of the LLL, from which the proof of the symmetric version follows at once.

Theorem 1 (General Lovász Local Lemma)**.** *Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ be a finite collection of events over an arbitrary probability space. Let $G = (V, E)$ be their dependency graph. If there exists an assignment of reals $x : \mathcal{A} \to [0, 1)$ such that*

$$\text{For all } i \in \{1, \ldots, m\} : \quad \mathbb{P}(A_i) \leqslant x(A_i) \prod_{A_j \in N(A_i)} (1 - x(A_j)),$$

then

$$\mathbb{P}\left(\bigcap_{i=1}^{m} A_i^c\right) \geqslant \prod_{i=1}^{m} (1 - x(A_i)) > 0,$$

and thus the bad events can be avoided altogether with non-zero probability.

We can think of $x(A_i)$ as the actual probability of event A_i in the absence of dependence, and the term $\prod_{A_j \in N(A_i)}(1 - x(A_j))$ as a "penalty" that event A_i pays due to its dependence on other events. Thus, the lighter the dependence of A_i on other events, the higher the probability of occurrence it can have and still be avoided. We note that Spencer's sufficient condition $ep(d + 1) \leqslant 1$ is a spacial case

of the theorem above; if $ep(d+1) \leqslant 1$, where $\mathbb{P}(A_i) \leqslant p$ for every i, then

$$\mathbb{P}(A_i) \leqslant p \leqslant \frac{1}{(d+1)e} \leqslant \frac{1}{d+1}\left(\frac{d}{d+1}\right)^d = \frac{1}{d+1}\prod_{i=1}^{d}\left(\frac{d}{d+1}\right) = \frac{1}{d+1}\prod_{i=1}^{d}\left(1-\frac{1}{d+1}\right)$$
$$\leqslant x(A_i)\prod_{A_j \in N(A_i)}(1-x(A_j))$$

when setting $x(A_i) = 1/(d+1)$ for every i. Thus setting $x(A_i) = 1/(d+1)$ in the asymmetric condition ensures that $ep(d+1) \leqslant 1$ holds. Note that the last inequality follows because $|N(A_i)| \leqslant d$ and $d/(d+1) < 1$ for any $d \geqslant 1$. We have also used the fact that $\left(\frac{d}{d+1}\right)^d \geqslant \frac{1}{e}$. This follows because

$$\left(\frac{d+1}{d}\right)^d = \left(1+\frac{1}{d}\right)^d \leqslant e.$$

We extend the definitions thus presented to random variables. We will assume that we are given a finite family of *mutually independent* random variables, $\mathcal{P}$, over a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Every random variable $P \in \mathcal{P}$, $P : \Omega \to \mathbb{R}$ assumes real values in a finite set, say V_P, $|V_P| < \infty$. We will assume that our events A_i are *determined* by a subset $\mathcal{S} \subseteq \mathcal{P}$. That is, $A_i \subseteq \bigcup_{S \in \mathcal{S}}\{\omega : S(\omega) = v_S\}$ for some evaluation of all the random variables in $\mathcal{S}$. We say that every such evaluation of the random variables *violates* event A_i, to emphasize that A_i is a bad event that we want to avoid, and an evaluation that determines A_i is undesirable. In the language of measure theory, we say that A_i is $\sigma(\mathcal{S})$ measurable; that is, $A_i \subseteq \sigma(\mathcal{S})$, where

$$\sigma(\mathcal{S}) \equiv \sigma(S^{-1}(v_S) : S \in \mathcal{S} \text{ and } v_S \in V_S) \subseteq \mathcal{F}$$

is the smallest σ-field that is generated by A_i. Such σ-field exists, and is in fact the intersection of all σ-fields that contain A_i (and σ-fields are closed under intersections). Thus it is necessary that there exists a minimal unique set $\mathcal{S} \subseteq \mathcal{P}$ that determines A_i.

We shall denote the set of variables that determine every event A_i as $\texttt{vbl}(A_i)$ for all $i \in \{1, \ldots, m\}$. This set is given to the algorithm for every event. We extend our notion of the dependency graph to random variables as well. The set of vertices is the set of events $\mathcal{A}$, but an edge exists between two events A_i and A_j iff $\texttt{vbl}(A_i) \cap \texttt{vbl}(A_j) \neq \emptyset$. This notion of independence is equivalent to that in the original definition, because if $\texttt{vbl}(A_i) \cap \texttt{vbl}(A_j) = \emptyset$, then setting the value of any random variable in $\texttt{vbl}(A_i)$ does not change the evaluation of any random variable in $\texttt{vbl}(A_j)$. Then it follows that $\sigma(\texttt{vbl}(A_i))$ and $\sigma(\texttt{vbl}(A_j))$ are independent, and thus for every $B_1 \in \sigma(\texttt{vbl}(A_j))$ and every $B_2 \in \sigma(\texttt{vbl}(A_j))$, we have $\mathbb{P}(B_1 \cap B_2) = \mathbb{P}(B_1)\mathbb{P}(B_2)$. But since $A_i \subseteq \sigma(\texttt{vbl}(A_i))$ and $A_j \subseteq \sigma(\texttt{vbl}(A_j))$, the above notion of independence for σ-fields implies that $\mathbb{P}(A_i \cap A_j) = \mathbb{P}(A_i)\mathbb{P}(A_j)$ whenever $\texttt{vbl}(A_i) \cap \texttt{vbl}(A_j) = \emptyset$.

Now given the set of events $\mathcal{A}$ and the set of random variables $\mathcal{P}$, if the events satisfy the conditions of the general LLL as stated in Theorem 1, then the following simple randomized algorithm efficiently finds an evaluation of the random variables in $\mathcal{P}$ such that none of the events is violated, that is, $\mathbb{P}\left(\bigcap_{i=1}^{m} A_i^c\right) > 0$. Initially a value for every random variable $P \in \mathcal{P}$ is sampled, each according to its distribution. Then if there are still violated events, then a violated event is chosen arbitrarily,

and a value is sampled for every variable that determines the event. The last process continues until there are no more violated events. We report the algorithm in Algorithm 1.

Algorithm 1: LLLConstruct$(\mathcal{P}, \mathcal{A})$

for *every* $P \in \mathcal{P}$ **do**
    Sample v_P from V_P and set $P \leftarrow v_P$
end for
while *there are violated events* **do**
    Pick a violated event A
    for *every* $P \in \texttt{vbl}(A)$ **do**
        Sample v_P from V_P and set $P \leftarrow v_P$
    end for
end while
return the final assignments $(v_P)_{P \in \mathcal{P}}$

The algorithm therefore finds an assignment of reals $x : \mathcal{A} \to \mathbb{R}$ such that $\mathbb{P}\left(\bigcap_{i=1}^{m} A_i^c\right) \geqslant \prod_{i=1}^{m}(1 - x(A_i)) > 0$ and, moreover, the algorithm terminates (if, again, the set of input events meet the conditions of the LLL). The major claim is

Claim 1. *The* expected *number of times an event, say $A_i \in \mathcal{A}$, might be resampled (and thus violated) by the Moser-Tardos algorithm is at most $\frac{x(A_i)}{1-x(A_i)}$.*

When the algorithm terminates with a final assignment $P = v_P$ for every $P \in \mathcal{P}$, we can find the set $\{\bigcap_{i=1}^{m} A_i^c\}$ by computing the set $\{\omega : P(\omega) = v_P\}$, and thus recover $x(A_i)$ for every $i \in \{1, \ldots, m\}$.

Next we analyze the algorithm and prove that it terminates as claimed above.

2.2 Analysis of the Moser-Tardos Algorithm

The analysis is based on the following idea: for every violated event that the algorithm resamples, or "fixes", we build a tree rooted at the thus resampled event. At every resampling step, this tree serves as a witness, or justification, for why the algorithm needed to resample that event. Moreover, at every sampling step, we will see that the witness tree constructed includes all the nodes that have been resampled so far, in reverse depth order. Thus the length of the tree at every resampling step will be upper bounded by the number of events violated and resampled so far. Our task is to bound the expected length of the tree, which implies bounding the expected number of resampling steps that the algorithm performs. Towards the latter goal, we will focus our efforts on bounding the probability that a witness tree occurs.

At every step the algorithm resamples a violated event, we keep a record of the events that the algorithm resampled so far. We call this list of events the *log of execution*. Formally, it is a function $C : \mathbb{N} \to \mathcal{A}$ that maps every step of the algorithm to the event in $\mathcal{A}$ that the algorithm has resampled at that particular step (note that C is a partial function; it is not defined on all $\mathbb{N}$ because the algorithm does terminate !)

At every resampling step t, the witness tree mentioned above is formally a finite tree $\texttt{wtree}(t)$ rooted at $C(t)$, whose vertices are subsets of $\mathcal{A}$, that is, $V(\texttt{wtree}(t)) \subseteq$

$\mathcal{A}$. At resampling step t, $\texttt{wtree}(t)$ is constructed inductively as follows. At $t = 1$, $\texttt{wtree}(1)$ consists of $C(1)$ as the sole vertex. At $t \geqslant 2$, given the log C until time t, create the root $C(t)$. Then consider the log C in reverse order, i.e., from $i = t - 1$ back to $i = 1$. We will say that $C(i)$ *overlaps* with a node $A_j \in V(\texttt{wtree}(t))$ if either $C(i) = A_j$, or there is an edge in the dependency graph G of $\mathcal{A}$ between $C(i)$ and A_j (that is, $\texttt{vbl}(C(i)) \cap \texttt{vbl}(A_j) \neq \emptyset$). Put equivalently, $C(i) \in N^+(A_j)$. For $i = t - 1, \ldots, 1$, denote as $\texttt{wtree}^i(t)$ the partial tree created after examining $C(i)$. Thus $\texttt{wtree}^t(t)$ consists of $C(t)$ solely, and $\texttt{wtree}(t) = \texttt{wtree}^1(t)$ (so constructing $\texttt{wtree}(t)$ involves growing a tree rooted at $C(t)$ by examining $C(i), C(i-1), \ldots, C(1)$, resulting in a possibly bigger partial tree at each step i, depending on whether or not $C(i)$ is attached to $\texttt{wtree}^{i+1}(t)$). If $C(i)$ overlaps with any nodes in the partial tree $\texttt{wtree}^{i+1}(t)$, then among all the nodes in $\texttt{wtree}^{i+1}(t)$ with which $C(i)$ overlaps, attach $C(i)$ to the node in $\texttt{wtree}^{i+1}(t)$ that is furthest from the root $C(t)$ (i.e., has the greatest depth; this is crucial) to get $\texttt{wtree}^i(t)$. If $C(i)$ does not overlap with any nodes in the partial tree $\texttt{wtree}^{i+1}(t)$, then simply skip $C(i)$.

The witness tree $\texttt{wtree}(t)$ has the following interesting properties.

Proposition 1. *Given log C at time t, the witness tree* $\texttt{wtree}(\text{t})$ *has the following properties*

(i) *If $r < s$, A_r is added to* $\texttt{wtree}(\text{t})$ *at time r and A_s added to* $\texttt{wtree}(\text{t})$ *at time s, and A_r and A_s overlap, then the depth of A_r in* $\texttt{wtree}(\text{t})$ *is greater than the depth of A_s,*

(ii) *All the vertices at the same level in* $\texttt{wtree}(\text{t})$ *are independent, and*

(iii) *All trees up to time t are different;* $\texttt{wtree}(1) \neq \texttt{wtree}(2) \neq \cdots \neq \texttt{wtree}(\text{t})$.

Proof. (i) follows from the construction of the $\texttt{wtree}(t)$: A_r is added to $\texttt{wtree}(t)$ after A_s, and is added to the vertex in $\texttt{wtree}^{r+1}(t)$ that is furthest from $C(t)$. So if A_r and A_s overlap, then it is attached to A_s as a child or to a deeper node in the tree with which it overlaps.
(ii) follows directly from (i).
For (iii), assume that $\texttt{wtree}(r) = \texttt{wtree}(s)$ for times $r < s$. Then $\texttt{wtree}(r)$ and $\texttt{wtree}(s)$ have the same root, that is, $C(r) = C(s) = A$ for some $A \in \mathcal{A}$. Consider the construction of $\texttt{wtree}(s)$. For every $i = s - 1, \ldots, 1$, if $C(i) = C(s)$, then $C(i)$ always appears as a node in $\texttt{wtree}(s)$. Thus the number of times $C(s)$ appears as a node in $\texttt{wtree}(s)$ is exactly the number of occurrences of $C(s)$ in the log C up to time s. Call this number $n_A(s)$. Thus all the nodes in the log C for $i = r, \ldots, 1$ with $C(i) = C(r)$ will appear in $\texttt{wtree}(r)$, and of course they all appear in $\texttt{wtree}(s)$, so those nodes are a subset of $V(\texttt{wtree}(s))$. In fact, they are a proper subset, because $n_A(s)$ is at least $n_A(r)+1$, because A appeared at least once after time r, specifically at time s, and thus it is not a node in $\texttt{wtree}(r)$. Therefore $\texttt{wtree}(r)$ and $\texttt{wtree}(s)$ contain different sets of vertices and thus $\texttt{wtree}(r) \neq \texttt{wtree}(s)$. □

Let $\texttt{wtree}$ be a witness tree. For a given log C, we say that witness tree $\texttt{wtree}$ occurs in the log C if there exists $t \in \mathbb{N}$ such that $\texttt{wtree} = \texttt{wtree}(t)$. With the properties above of the witness tree in hand, we are in a position to prove the following

Theorem 2. *For a log C, the probability that a witness tree* $\texttt{wtree}$ *occurs in C is* $\prod_{A_i \in V(\texttt{wtree})} \mathbb{P}(A_i)$.

Proof. The proof proceeds by a coupling argument. Fix the random source that the algorithm uses when sampling observations of each random variable $P \in \mathcal{P}$. Assume that every $P \in \mathcal{P}$ offers an infinite sequence of independent observations (samples) from the elements in V_P, arranged in some sequence $v_P(0), v_P(1), \ldots$, where $v_P(k) \in V_P$ for every $k \in \mathbb{Z}_+$ (recall that V_P is finite)[1]. Whenever the algorithm needs a new sample from a random variable, say P, either in the initial assignment or when an event is violated, it chooses the next unused sample in the sequence $v_P(0), v_P(1), \ldots$. Now define the following procedure, called `wtree`-check: starting from the bottom of the tree (i.e., reverse depth order), visit every node $A_i \in V(\texttt{wtree})$ in this order. For every $A_i \in V(\texttt{wtree})$ in this (deterministic) walk, sample an evaluation of the random variables in $\texttt{vbl}(A_i)$, independently of all previous evaluations, and according to A_i's distribution. Check if this evaluation violates (produces) A_i. Stop the `wtree`-check the first time a failure occurs, that is, at the first event that is not violated by the sampling performed by `wtree`-check. If all nodes are violated when we finish the `wtree`-check (which happens when we sample the root), then we say that the `wtree`-check passes. Assume that the `wtree`-check uses the fixed, infinite random source defined above when it chooses a sample for any random variable P. Since the `wtree`-check samples every event independently of all previous evaluations, it follows that the probability that the `wtree`-check passes is exactly $\prod_{A_i \in V(\texttt{wtree})} \mathbb{P}(A_i)$. We will show that `wtree` occurs in C iff the `wtree`-check passes when both the algorithm and the `wtree`-check use the same fixed random source.

Consider what happens when the algorithm resamples an event in $\mathcal{A}$, say A_i. Consider all the events that overlap with A_i that have been resampled (violated) before A_i. Denote this set of events as $\mathcal{E}$. All such events are in the tree, and are at depth that is strictly greater than that of A_i in `wtree` (by Proposition 1). Of course, A_i shares at least one variable with every event in $\mathcal{E}$; suppose that A_i shares a variable, say P, with all those events. Then, when the algorithm considers event A_i, the value of P is precisely $v_P(|\mathcal{E}|)$, and the algorithm has sampled $|\mathcal{E}| + 1$ values for P (one for the initial assignment plus $|\mathcal{E}|$ values, a value per event in $\mathcal{E}$). Now consider the `wtree`-check: when, during its upward walk, it examines event A_i, the `wtree`-check has already sampled the set $\mathcal{E}$, because the `wtree`-check moves bottom-up. Moreover, when the `wtree`-check has reached A_i, then the check has passed for every $E \in \mathcal{E}$, meaning that all the events in $\mathcal{E}$ have been violated. Most importantly, the value of P when the `wtree`-check considers event A_i is exactly $v_P(|\mathcal{E}|)$ and, since both the algorithm and the `wtree`-check have the same values for P, then the `wtree`-check will find that A_i is violated. Thus on the same random sequence, the `wtree`-check passes if, and only if, `wtree` exists in C, so the probability that the `wtree` exists in C is exactly the probability that the `wtree`-check passes. This concludes the proof. □

Proposition (1) part iii tells us even more; it implies that the number of witness trees rooted at a certain node, say A_i, is exactly the number of times that A_i appears in the log (because trees are distinct even if they have the same root), which in turn is the number of times that A_i has been resampled. Now that we know the probability of occurrence of each tree, the expected number of trees rooted at A_i (and thus the

[1]From an implementation perspective, fixing the random sequence for random variables amounts to starting the random number generators with the same seed .

number of times event A_i has been resampled) can be easily computed by summing the probabilities above. Let $\mathcal{T}_{A_i}$ be the set of all distinct witness trees rooted at A_i. Let R_{A_i} be a random variable that indicates the number of times that A_i has been resampled throughout the course of execution of the algorithm. Then R_{A_i} counts the number of witness trees rooted at A_i in C;

$$R_{A_i} = \sum_{\texttt{wtree} \in \mathcal{T}_{A_i}} 1_{\{\texttt{wtree} \text{ occurs in } C\}}.$$

Thus

$$\begin{aligned}
\mathbb{E}(R_{A_i}) &= \sum_{\texttt{wtree} \in \mathcal{T}_{A_i}} \mathbb{P}(\texttt{wtree} \text{ occurs in } C) \\
&= \sum_{\texttt{wtree} \in \mathcal{T}_{A_i}} \prod_{A_j \in V(\texttt{wtree})} \mathbb{P}(A_j) \\
&\leqslant \sum_{\texttt{wtree} \in \mathcal{T}_{A_i}} \prod_{A_j \in V(\texttt{wtree})} x(A_j) \prod_{A_k \in N(A_j)} (1 - x(A_k)), \qquad (4)
\end{aligned}$$

where the last inequality follows from Theorem 1. To prove Claim 1 above, that $\mathbb{E}(R_{A_i}) \leqslant x(A_i)/(1-x(A_i))$, we need to write the RHS of Eqn. (4) in terms of $x(A_i)$ only. To do so, we find an upper bound on $\prod_{A_j \in V(\texttt{wtree})} x(A_j) \prod_{A_k \in N(A_j)} (1 - x(A_k))$ in terms of $x(A_i)$, by relating witness tree creation to a Branching process[2]

2.2.1 Branching Processes

In brief, a Branching process is a Markov chain, where an individual gives birth to immediate children, which we call the *family* of that individual. The process starts at a root parent that gives birth to a random number of children, each is born randomly according to its own distribution (given its parent), and independently of other siblings. Those children form the first generation. Then, independently of each other, every child gives birth to a (random) number of individuals, and so on. Therefore, a tree is formed by this process, where the individuals at the nth level of the tree are called the nth generation. The crucial point is that, given a parent, its family is independent (or grows independently) of all other families in the tree (here is where the Markov property kicks in.)

Next we give a brief overview of the most basic of branching processes : the Single type Galton-Watson process. Let ζ_i^n, $i \geqslant 1$, $n \geqslant 1$, be i.i.d, non-negative integer-valued random variables. Every ζ_i^n is the number of immediate children that the ith individual in the $(n-1)$st generation gives birth to in the nth generation. Define the sequence of random variables Z_n, $n \geqslant 1$, where Z_n is the number of individuals in the nth generation, and $Z_0 = 1$. Then naturally,

$$Z_{n+1} = \begin{cases} \zeta_1^{n+1} + \ldots + \zeta_{Z_n}^{n+1} & \text{if } Z_n > 0, \\ 0 & \text{if } Z_n = 0. \end{cases}$$

The *offspring distribution* of the ith individual in the $(n-1)$st generation and gives birth to children in the nth generation is $\mathbb{P}(\zeta_i^n = k)$ for any $k \in \mathbb{Z}_+$. The

[2]Some authors use the term Galton-Watson process to refer to Branching processes.

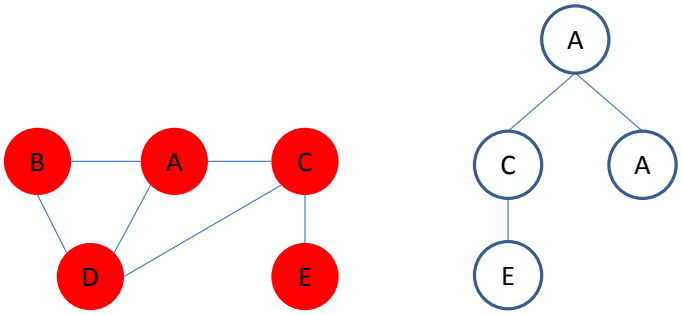
extinction probability is the probability that the branching process dies out, that is, $\lim_{n\to\infty} \mathbb{P}(Z_n = 0)$. Thus the process dies out if $\lim_{n\to\infty} \mathbb{P}(Z_n = 0) = 1$. Let $\mathcal{F}_n = \sigma(\zeta_m^i : m \in \{1, \ldots, n\}, i \geqslant 1)$. If $\mu = \mathbb{E}(\zeta_i^n)$ (recall the ζ_i^ns are i.i.d and thus have the same mean), then it is known that Z_n/μ^n is an $(\mathcal{F}_n)$-martingale. Moreover, Z_n/μ^n is L^1-Bounded, that is, $\sup_n \mathbb{E}(|Z_n/\mu^n|) < \infty$. Therefore, Doob's Forward Martingale Convergence Theorem applies, and we have that Z_n/μ^n converges to some limit almost surely. Specifically, if $\mu < 1$, then it is always the case that $Z_n/\mu^n \to 0$ almost surely, and that $\lim_{n\to\infty} \mathbb{P}(Z_n > 0) = 0$, so the process always dies in this case (the process is called subcritical when $\mu < 1$). In words, if an individual gives birth to less than one child on average, then the process will eventually die out. We invite the reader to examine Durrett [2], chap. 5, for neat proofs of the results above.

Now given the background above, we get back to the task of bounding Eqn. (4) in terms of $x(A_i)$ only. Let $\texttt{wtree} \in \mathcal{T}_{A_i}$. For each event $A \in \mathcal{A}$, think of $x(A)$ as a conditional probability: it is the probability that A exists as a vertex in $\texttt{wtree}$ as a child of any vertex in $N^+(A)$ in the dependency graph, that is, given that a node in $N^+(A)$ exists in $\texttt{wtree}$, and that this node is the immediate parent of A. Now consider the following Galton-Watson process for generating a tree rooted at A_i, call $\texttt{wtree}_{\text{GW}}$. First, designated A_i as a root. Then for each $A_j \in N^+(A_i)$, independently either attach A_j to A_i with probability $x(A_j)$, or skip A_j with probability $1 - x(A_j)$. Repeat the process above for every node you add in $\texttt{wtree}_{\text{GW}}$. This process will either die out eventually, or will continue forever (extinction depends on means as we outlined above, which in turn depend on node probabilities). Our question will be: what is the probability that $\texttt{wtree}_{\text{GW}} = \texttt{wtree}$?

Comparing with our overview above, we see that the random variables ζ_i^n in our witness tree generation branching process are not identically distributed; this is because reproduction depends on the parent's type. Different events have different neighbors in the dependency graph, and so events are of different types, and we have a *multitype* Galton-Watson process. Thus to describe the offspring distribution of a node, say B, which occurs as the ith individual in the nth generation, it is not enough to specify $\mathbb{P}(\zeta_i^{n+1} = k)$ for every k. Rather, if $N^+(B) = \{B_1, \ldots, B_d\}$ is the inclusive neighborhood of B in the dependency graph G of $\mathcal{A}$, then the offspring distribution is the joint distribution $\mathbb{P}(\zeta_{i,B_1}^{n+1} = k_1, \ldots, \zeta_{i,B_d}^{n+1} = k_d)$ for all $(k_1, \ldots, k_d) \in \mathbb{Z}_+^d$. This expression is just another way to encode families; we will interpret it as the probability that this family occurs. We point out that the results presented above for single type Galton-Watson processes carry over to multitype Galton-Watson processes. We refer the reader to Athreya and Ney [1] for a treatment of multitype Galton-Watson processes.

In our tree generation process, children of a certain node are chosen independently of each other, one at a time, so $\mathbb{P}(\zeta_{i,B_1}^{n+1} = k_1, \ldots, \zeta_{i,B_d}^{n+1} = k_d) = \prod_{j=1}^d \mathbb{P}(\zeta_{i,B_j}^{n+1} = k_j)$, where $\mathbb{P}(\zeta_{i,B_j}^{n+1} = k_j) = \mathbb{P}(\zeta_{i,B_j}^{n+1} = 1)^{k_j} = x(B_j)^{k_j}$ if $k_j \geqslant 1$, and $\mathbb{P}(\zeta_{i,B_j}^{n+1} = 0) = 1 - x(B_j)$. Now computing the probability of the event $\{\texttt{wtree}_{\text{GW}} = \texttt{wtree}\}$ becomes a simple task. We illustrate the process by means of an example.

Example. Consider the witness tree in Figure 3. Assume that this tree occurs in the log C when the Moser-Tardos algorithm resamples some event $A \in \mathcal{A}$. Call this tree $\texttt{wtree}$, where $\texttt{wtree} \in \mathcal{T}_A$. The probability that our described Galton-Watson process produces the same tree when starting at node A is the probability



(a) Dependency Graph

(b) Witness tree rooted at A Generated by some log C

Figure 3: Dependency Graph and Witness Tree for Galton-Watson Process Tree Generation Example

that every family in $\texttt{wtree}$ occurs, which is given by

$$\begin{aligned}\mathbb{P}(\texttt{wtree}_{\text{GW}} = \texttt{wtree}) = \mathbb{P}((\zeta^1_{1,C} = 1, \zeta^1_{1,A} = 1, \zeta^1_{1,B} = 0, \zeta^1_{1,D} = 0) \cap \\ (\zeta^2_{1,D} = 1, \zeta^2_{1,E} = 0, \zeta^2_{1,C} = 0, \zeta^2_{1,A} = 0) \cap \\ (\zeta^2_{2,A} = 0, \zeta^2_{2,B} = 0, \zeta^2_{2,C} = 0, \zeta^2_{2,D} = 0) \cap \\ (\zeta^3_{1,C} = 0, \zeta^3_{1,E} = 0)).\end{aligned}$$

But families grow independently given their parents (the Markovian property). So

$$\begin{aligned}\mathbb{P}(\texttt{wtree}_{\text{GW}} = \texttt{wtree}) = &\mathbb{P}(\zeta^1_{1,C} = 1, \zeta^1_{1,A} = 1, \zeta^1_{1,B} = 0, \zeta^1_{1,D} = 0) \times \\ &\mathbb{P}(\zeta^2_{1,D} = 1, \zeta^2_{1,E} = 0, \zeta^2_{1,C} = 0, \zeta^2_{1,A} = 0) \times \\ &\mathbb{P}(\zeta^2_{2,A} = 0, \zeta^2_{2,B} = 0, \zeta^2_{2,C} = 0, \zeta^2_{2,D} = 0) \times \\ &\mathbb{P}(\zeta^3_{1,C} = 0, \zeta^3_{1,E} = 0).\end{aligned}$$

And by our Galton-Watson process,

$$\begin{aligned}\mathbb{P}(\texttt{wtree}_{\text{GW}} = \texttt{wtree}) = &x(C)x(A)(1-x(B))(1-x(D)) \times \\ &x(E)(1-x(D))(1-x(C))(1-x(A)) \times \\ &(1-x(A))(1-x(B))(1-x(C))(1-x(D)) \times \\ &(1-x(C))(1-x(E)).\end{aligned}$$

Rearranging the terms in the last equation we get

$$\begin{aligned}\mathbb{P}(\texttt{wtree}_{\text{GW}} = \texttt{wtree}) = &x(A)[(1-x(B))(1-x(D))] \times \\ &x(C)[(1-x(A))(1-x(C))(1-x(D))] \times \\ &1[(1-x(A))(1-x(B))(1-x(C))(1-x(D))] \times \quad (5) \\ &x(E)[(1-x(C))(1-x(E))].\end{aligned}$$

Note that the term (5) in the equation above corresponds to the events that do not exist as children of event A in the second level of the tree ($N^+(A) = \{A, B, C, D\}$). This term can be written as $\frac{1}{x(A)}x(A)[(1-x(A))(1-x(B))(1-x(C))(1-x(D))]$.

Therefore, if for every $A \in \mathcal{A}$ we let W_A denote the subset of vertices in $N^+(A)$ that do not occur as children of node A in $\mathtt{wtree}_{\mathrm{GW}}$, then the equation above can be written as

$$\mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) = \frac{1}{x(A)} \prod_{A_j \in V(\mathtt{wtree}_{\mathrm{GW}})} x(A_j) \prod_{A_k \in W_{A_j}} (1 - x(A_k)).$$

In fact, the result in the example above is a general one and is not a mere coincidence: for any set of events $\mathcal{A} = \{A_1, \ldots, A_m\}$ satisfying the conditions of the LLL as in Theorem 1, for every $A_i \in \mathcal{A}$ and every $\mathtt{wtree} \in \mathcal{T}_{A_i}$, the probability that the tree $\mathtt{wtree}_{\mathrm{GW}}$ produced by our described Galton-Watson process when started at A_i is identical to $\mathtt{wtree}$ is

$$\mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) = \frac{1}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \prod_{A_k \in W_{A_j}} (1 - x(A_k)).$$

Now that we have this result in hand, we are in business to prove the following

Lemma 1. *Let* $\mathtt{wtree}$ *be a fixed witness tree rooted at* A_i*. The probability that the witness tree* $\mathtt{wtree}_{GW}$ *generated by the described Galton-Watson, when started at* A_i*, is identical to* $\mathtt{wtree}$ *is*

$$\mathbb{P}(\mathtt{wtree}_{GW} = \mathtt{wtree}) = \frac{1 - x(A_i)}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \prod_{A_k \in N(A_j)} (1 - x(A_k)).$$

Proof. From above, we have

$$\begin{aligned}
\mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) &= \frac{1}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \prod_{A_k \in W_{A_j}} (1 - x(A_k)) \\
&= \frac{1}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \frac{\prod_{A_k \in N^+(A_j)} (1 - x(A_k))}{\prod_{A_k \in N^+(A_j) \setminus W_{A_j}} (1 - x(A_k))} \\
&= \frac{1}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} \frac{1}{\prod_{A_k \in N^+(A_j) \setminus W_{A_j}} (1 - x(A_k))} x(A_j) \prod_{A_k \in N^+(A_j)} (1 - x(A_k)).
\end{aligned}$$

Looking at the denominator of the last equation, we see that for every node $A_j \in V(\mathtt{wtree})$, we are multiplying the probabilities that its immediate children in $\mathtt{wtree}$ do not occur in $\mathtt{wtree}_{\mathrm{GW}}$, so the denominator is accounting for every node in $\mathtt{wtree}$ except for the root A_i. Thus

$$\prod_{A_j \in V(\mathtt{wtree})} \frac{1}{\prod_{A_k \in N^+(A_j) \setminus W_{A_j}} (1 - x(A_k))} = \prod_{A_j \in V(\mathtt{wtree})} \frac{1 - x(A_i)}{1 - x(A_j)}.$$

So we have

$$\begin{aligned}
\mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) &= \frac{1 - x(A_i)}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} \frac{x(A_j)}{1 - x(A_j)} \prod_{A_k \in N^+(A_j)} (1 - x(A_k)) \\
&= \frac{1 - x(A_i)}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \frac{\prod_{A_k \in N^+(A_j)} (1 - x(A_k))}{(1 - x(A_j))} \\
&= \frac{1 - x(A_i)}{x(A_i)} \prod_{A_j \in V(\mathtt{wtree})} x(A_j) \prod_{A_k \in N(A_j)} (1 - x(A_k)). \qquad (6)
\end{aligned}$$

□

Now the proof of Claim 1 is a corollary of what we have proved so far. Recall Eqn. (4). Then

$$\begin{aligned}\mathbb{E}(R_{A_i}) &\leqslant \sum_{\mathtt{wtree}\in\mathcal{T}_{A_i}} \prod_{A_j\in V(\mathtt{wtree})} x(A_j) \prod_{A_k\in N(A_j)} (1-x(A_k)) \\ &\leqslant \sum_{\mathtt{wtree}\in\mathcal{T}_{A_i}} \frac{x(A_i)}{1-x(A_i)} \mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) \qquad \text{[by Eqn. (6)]} \\ &= \frac{x(A_i)}{1-x(A_i)} \sum_{\mathtt{wtree}\in\mathcal{T}_{A_i}} \mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree})\end{aligned}$$

Now the situation is the following: the tree $\mathtt{wtree}_{\mathrm{GW}}$ generated by the Galton-Watson process might grow infinite. Even if the Galton-Watson halts, then it might not generate any tree in $\mathcal{T}_{A_i}$ when started at A_i. Moreover, a single run of our Galton-Watson process produces one tree $\mathtt{wtree}_{\mathrm{GW}}$, which can therefore be identical to at most one tree in $\mathcal{T}_{A_i}$. Thus $\sum_{\mathtt{wtree}\in\mathcal{T}_{A_i}} \mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) \leqslant 1$ and we have

$$\mathbb{E}(R_{A_i}) \leqslant \frac{x(A_i)}{1-x(A_i)} \sum_{\mathtt{wtree}\in\mathcal{T}_{A_i}} \mathbb{P}(\mathtt{wtree}_{\mathrm{GW}} = \mathtt{wtree}) \leqslant \frac{x(A_i)}{1-x(A_i)}.$$

In conclusion, the total number of resampling (event fixing) steps that the Moser-Tardos algorithm performs when the set of bad events is $\mathcal{A} = \{A_1, \ldots, A_m\}$ is, on average, $\sum_{i=1}^{m} \frac{x(A_i)}{1-x(A_i)}$.

2.3 The Moser-Tardos LLL Algorithm Admits Parallelism

Recall that in Algorithm 1, we pick a violated event to fix *one at a time*. We can do better, however. Note that the events that do not overlap in the dependency graph form an *independent set*, in the sense that altering the assignment of any variable in the set of variables of any event in this independent set does not change the assignment of variables of other events. This is simply because independent events do not have any variables in common. Therefore, instead of picking one violated event at a time, the algorithm (greedily) picks a maximal independent set of violated events in the dependency graph G at every resampling step. For the maximal independent set of violated events that the algorithm picks, call $\mathtt{IS} \subseteq \mathcal{A}$, it samples an assignment for every variable $P \in \bigcup_{A\in\mathtt{IS}} \mathtt{vbl}(A)$ independently at random. Note that there are exactly $\sum_{A\in\mathtt{IS}} |\mathtt{vbl}(A)|$ variables to set at every resampling step, since the sets $\mathtt{vbl}(A)$ are disjoint in $\mathtt{IS}$.

We do not delve into the analysis of the parallel version of the algorithm, but we point out that in order to achieve a logarithmic expected number of resampling steps, the condition that

$$\textit{For all } i \in \{1,\ldots,m\}: \quad \mathbb{P}(A_i) \leqslant x(A_i) \prod_{A_j\in N(A_i)} (1-x(A_j)),$$

as stated in Theorem 1 had to be weakened to

$$\textit{For all } i \in \{1,\ldots,m\}: \quad \mathbb{P}(A_i) \leqslant (1-\epsilon)x(A_i) \prod_{A_j\in N(A_i)} (1-x(A_j)),$$

for some $\epsilon > 0$. If the set of input events satisfy this condition, then the number of resampling steps is, on average, $O\left(\frac{1}{\epsilon}\log\left(\sum_{i=1}^{m}\frac{x(A_i)}{1-x(A_i)}\right)\right)$.

3 The LLL and Packet Routing and Job Scheduling

In the sequel, we present a non-trivial application of the LLL to Packet Routing and thus Job Scheduling, the latter being an active research area in Operations Research. We present the results communicated in the beautiful article of Leighton, Maggs, and Rao [4].

Simply put, given a set of elements, called packets, and a network, the packet routing problem is concerned with moving all the input packets from their initial locations (origins) to their final destinations through the given network. The network we refer to is defined in the traditional network-theoretic sense: It is a digraph $D = (N, A)$, where N is a set of nodes, and A is a set of arcs (directed edges)[3], $m := |A|$. Thus packets start their journey at specified origin nodes, and they are required to traverse arcs, at most one arc at every time step, until they reach their final destinations. We assume that there is a global clock to which all the nodes are synchronized; therefore, a packet traverses an edge only at clock ticks (time is discrete in our setting) and in its entirety (packets are atomic objects). We insist that at most one packet can traverse an edge at every clock tick (this requirement will be an essential property of the desired schedule of the packets).

When a packet traverses an edge, it is placed in a queue at the end of that edge. Such queues will have specific finite sizes, and thus a packet is allowed to traverse an edge only if the edge queue is not full. Specifically, a packet that is ready to traverse an edge at the next time step is at the head of the queue of an edge adjacent to the one it wants to traverse, and a packet queued must wait for all the packets ahead of it in the queue to be transferred before it becomes ready to traverse an adjacent edge (so queues can be though of as belonging to the nodes themselves.)

We define a *schedule* of the packets as the series of decisions of which packet to move across every edge at every time step. In particular, a schedule is a mapping between time steps and the packets that traverse every edge at every time step. Formally, denote the set of input packets (or jobs) as $\mathcal{J}$, where $n := |\mathcal{J}|$. For every $i \in \{1, \ldots, m\}$, packet J_i is identified by a pair (s_i, t_i), where $s_i, t_i \in A$, s_i is the source arc (origin), and t_i is the destination arc (terminal). We fix a labeling of the edges A of the underlying network D, say $(a_1, \ldots, a_m)$. Define a schedule snapshot vector $S = (J_1, \ldots, J_m)$, where the kth entry J_k is the packet that traverses edge a_k, if any, so either $J_k \in \mathcal{J}$, or $J_k = \emptyset$. Denote the set of all possible snapshots that can occur throughout the whole duration of any schedule of the packets as $\mathcal{S}$ (there are at most $(n+1)^m$ such vectors). Then a schedule is a partial injective function $\sigma : \mathbb{N} \to \mathcal{S}$ such that $\sigma(t) = S$ for some $S \in \mathcal{S}$.

Define the *makespan* of a certain schedule as the epoch, measured at $t = 0$, at which the last packet is delivered to its final destination. In other words, the makespan of a schedule is its length.

We distinguish between the *path selection problem* and the *routing* problem. In

[3]In what follows we shall use the terms "arc" and "edge" interchangeably.

the path selection problem, we are given the input packets with their origins and final destinations, along with the network, and the goal is to select paths for the packets so that packets are delivered to their final destinations, with some objective to optimize. In the routing problem, every packet has a specific path that it should traverse explicitly until it reaches its final destination, and the objective is to determine at every time step which packet traverses which edge. In this article we are concerned with the routing problem, and we aim at finding a schedule σ with as minimum a makespan as possible, while minimizing the maximum queue sizes needed to route the packets to their destinations. A *legitimate* schedule in our setting is one where at most one packet traverses an edge at a time. Therefore, if a schedule allows multiple packets, say x packets, to pass through an edge at a single clock tick, then this step can simulated by a legitimate schedule in x steps.

We will assume that every packet has a path P associated with it, and we will denote the set of all such input paths as $\mathcal{P}$, with $|\mathcal{P}| = |\mathcal{J}| = n$. A lower bound on the makespan of any legitimate schedule of the input packets is the length of the longest path in $\mathcal{P}$, since at least one packet in $\mathcal{J}$ must traverse that path. We call this quantity the *dilation*, d, of $\mathcal{P}$, where $d = \max_{P \in \mathcal{P}} |A(P)|$. A path P is subgraph of D, so we used the notation $A(P)$ to denote the set of arcs comprising P, where $A(P) \subseteq A$. Moreover, the maximum number of packets that can simultaneously use a single edge any time during the schedule is the *congestion*, c, of $\mathcal{P}$. We define the latter as $c = \max_{a \in A} \max |\{P \in \mathcal{P} : a \in A(P)\}|$. So the congestion is as well a lower bound on the makespan of any legitimate schedule of the packets; at least one packet will wait in a queue for c time steps before it traverses that edge (and a packet must traverse at least one edge). Thus, a lower bound on the makespan of any schedule of the packets is $(c + d)$. We note that according to our definitions, not all vectors in the set $\mathcal{S}$ we defined it above are valid.

A quite natural, though not legitimate, schedule is the greedy one, call σ_0: move *all* packets waiting at an edge queue across their desired edges. The non-legitimate algorithm we described will have a makespan of at most d, since a packet never waits for other packets on an edge. To turn this schedule to a legitimate one, we notice that if the congestion of $\mathcal{P}$ is c, then at most c packets will traverse any edge at any time instant by σ_0, so at most c packets will accumulate at a certain edge queue if we allow only one packet to traverse an edge at a time, thus a queue of size c suffices. We just simulate every step of the non-legitimate schedule: since queues are of size c, then a packet need to wait in a queue for at most $c - 1$ time steps before it becomes ready to traverse an edge in a legitimate schedule (and blocking due to full queues can never happen). Thus, we can simulate every step of the non-legitimate schedule in at most c steps, yielding a greedy legitimate schedule with a makespan of $O(cd)$.

With randomness, the authors achieved a much better upper bound on the makespan of any legitimate schedule. Remarkably, the upper bound of the makespan of the resulting schedule matches the lower bound, $O(c+d)$, with queue sizes of $O(1)$! In this article, however, we will content ourselves with a weaker result, that a schedule of makespan $(c + d)2^{O(\log^*(c+d))}$ and maximum queue size $2^{O(\log^*(c+d))} \log(c + d)$ exists, where $\log^*(x)$ is the *iterated logarithm* function of x. We explain this function in Appendix B. This result is not any less substantial than the stronger result, and the proof techniques used in the stronger result are extensions to the ones used in the weaker result. Interestingly, the bounds of both results are independent of

the number of input packets, n. The proofs of the main results do not explicitly construct the schedule, but rather employ the LLL to show the existence of such schedule. After we show that the schedule claimed above exists, we will outline how the Moser-Tardos algorithm can be applied to actually construct the schedule.

3.1 A $(c+d)2^{O(\log^*(c+d))}$ Routing Algorithm Exists

The algorithm is based on three main ideas. First, a random delay is assigned to every input packet, initially. Second, a non-legitimate, or unconstrained, greedy schedule, σ_0, is constructed, where every packet moves at every step, without ever stopping, until it reaches it final destination. Then successive refinements are applied to the greedy schedule, until at most a single packet is allowed to traverse an edge. Finally, the bad events are identified, where, roughly speaking, a bad event is the event that more than a certain number packets traverse a certain edge at any time (and so we have a bad event per edge). If the probability that all bad events do not take place simultaneously is strictly positive, then with positive probability, a set of packet delays exist such that it is highly unlikely that edges are highly congested, which yields an upper bound on both the schedule's makespan and edge queue sizes.

Now we use the ideas above to show the existence of a schedule with the claimed makespan and maximum queue size. We introduce the following notation. Define a T-frame to be a a sequence of T time steps. Let C denote the *frame congestion*, which is the maximum number of packets that can traverse any edge within time frame T. Note the frame congestion is time-related, whereas the input paths congestion, c, has no notion of time. Finally, Denote as R the relative congestion in a T-frame, which we define as $R = C/T$.

We will make the following assumption: every path $P \in \mathcal{P}$ is simple, in that no packet can use an edge more than once throughout its journey from the origin to the final destination. Next we present the fundamental Lemma that is going to be used in the proof of the main result.

Lemma 2. *For any set of packets whose paths are simple, and having congestion c and dilation d, there exists a schedule of makespan $O(c+d)$ in which packets never wait in edge queues, and in which the relative congestion R in a T-frame of length $\log d$ or greater is never greater than unity.*

Proof. First assign an initial delay to every packet. The delays are chosen uniformly and independently at random from the set $\{1, \ldots, \alpha d\}$; α is a constant to be determined. Recall our greedy schedule σ_0, where a packet never waits in a queue until it reaches its final destination. We refine σ_0 to a schedule σ_1, where a packet waits only in its initial queue for its randomly chosen delay, say x, then it moves, without ever waiting in any intermediate queue, until it reaches its final destination. Since at least one packet has a path of d edges, and packets cannot wait for more than αd time steps in their initial queues, it follows that the makespan of σ_1 is at most $(d + \alpha d)$. We would like to show that, if the delays are chosen as above, then there is a non-zero probability that the relative congestion in any T-frame of length at least $\log d$ is at most one. An application of the LLL will allow us to establish that such delays exist as follows. For every arc $a \in A$, define a bad event as: "more than T packets traverse arc a within some time frame of length T, for all frame lengths $T \geqslant \log d$." Thus showing that all those $|A| = m$ bad events do not take place simultaneously with non-zero probability implies that there exists a set of initial delays,

when chosen uniformly and independently at random, such that relative congestion in any T-frame of length at least $\log d$ is at most one.

We note that two bad events are dependent iff a packet passes through both the edges to which the bad events are associated. Thus, dependencies arise solely by delay assignments. At most c packets pass through any arc, and each of these packets can pass through at most d other arcs, so there cannot be more than cd dependent events. If we assume, wlog, that $c = d$, then the dependency parameter, b, is at most $b = cd = d^2$. Thus the maximum degree of any node in the dependency graph corresponding to our bad events is at most $b = d^2$.

Having identified dependencies, we proceed to compute an upper bound on the probability, p, of bad events. We will derive a uniform upper bound on the bad event probabilities so that we can apply the symmetric LLL. We observe the following.

(i) For edge a, define as success the event that a single packet traverses a during time frame T. A packet has one of αd delays assigned to it, independently of other packets, each with probability $1/\alpha d$. Since every packet's path is simple, a packet will never traverse an edge again once it passes through it. Thus, T delays can send the packet through a, only once, and so the probability that a packet passes through edge a during T is exactly $T/\alpha d$. We know that for any edge, at most $c = d$ packets can pass through that edge, and so, by our simple path assumption, there are at most $\binom{d}{k}$ combinations of k *distinct* packets that can pass through any edge at any time instant. Therefore, during a frame of length T, the probability that k packets pass through edge a is at most $\binom{d}{k}(T/\alpha d)^k(1 - T/\alpha d)^{d-k}$. If we let X_a be the number of packets that pass through an arc a during a T-frame, then X_a is a Binomial random variable, where $X_a \sim \text{BIN}(d, T/\alpha d)$, and $\mathbb{P}(X_a = k) = \binom{d}{k}(T/\alpha d)^k(1 - T/\alpha d)^{d-k}$;

(ii) If $T \geqslant d$, then $R = C/T \leqslant C/d \leqslant 1$, because $C \leqslant c = d$. So the bad events cannot occur for $T \geqslant d$, and we can therefore safely ignore frames of length greater than or equal to d. Thus, the frame sizes over which we would like to bound our probabilities becomes $\log d$ to d, and

(iii) The makespan of schedule σ_1 is at most $(1 + \alpha)d$, and thus for every frame duration T, time frames of duration T in σ_1 can start at any of $t = 0$ to $t = (1 + \alpha)d$, so there are at most $(1 + \alpha)d$ frames corresponding to every T.

From the observations above, we conclude that the probability that the bad event associated with edge a occurs is the probability, taken over all frame sizes $T = \log d$ to $T = d$ (observation (ii)), that the number of packets passing through edge a is greater than T during all time frames corresponding to every possible frame length T (observation (iii)), which we write as

$$p_a \leqslant \sum_{T=\log d}^{d} (1 + \alpha)d\mathbb{P}(X_a > T).$$

But from observation (ii), we have

$$\mathbb{P}(X_a > T) = \sum_{j=T+1}^{d} \binom{d}{j}(T/\alpha d)^j(1 - T/\alpha d)^{d-j}.$$

From the last expression, we note that $\mathbb{P}(X_a > T)$ does not depend on the particular edge a, so we get a uniform bound on the probability of bad events $p = p_a$ for all $a \in A$, which we write

$$p \leqslant \sum_{T=\log d}^{d} (1+\alpha)d \sum_{j=T+1}^{d} \binom{d}{j} (T/\alpha d)^j (1 - T/\alpha d)^{d-j}.$$

For sufficiently large, but fixed α, Shearer's LLL sufficient condition $epb \leqslant 1$ can be satisfied, which implies that all bad events can be avoided altogether with non-zero probability. Thus there exists a set of delays such that the relative congestion of any edge during any time frame of length $\log d$ or greater is less than one. Our lemma follows because the makespan of the schedule is $(1+\alpha)d \in O(d) = O(c+d)$ by our assumption that $c = d$. □

Note that, so far, we have not derived a bound on the maximum congestion for the whole duration of the schedule, we restricted our attention to $\log d$-sized frames and bounded the congestion during those frames, with high probability. Having shown the lemma above, we are in a position to prove our desired result.

Theorem 3. *For any set of packets whose paths are simple, have dilation d and congestion c, there exists a legitimate schedule of makespan $(c+d)2^{O(\log^*(c+d))}$ and maximum queue size $2^{O(\log^*(c+d))} \log(c+d)$.*

Proof. In order to write the bounds in terms of d only, we assume $c = d$ such that the makespan reads $d2^{O(\log^*(d))}$. The idea is to recursively divide the schedule σ_1 we produced in Lemma 2 and solve every smaller scheduling problem independently of others. Recall that the makespan of σ_1 is $(1+\alpha)d$, and that no more than $\log d$ packets can traverse any edge simultaneously in any frame of length $\log d$. The last property says that the congestion during a time frame of length $\log d$ is at most $\log d$, so we use it as follows: divide the makespan of σ_1 into frames of length $\log d$. Therefore, we get $(1+\alpha)d/\log d$, $\log d$ time frames.

We treat every $\log d$ frame as a separate scheduling problem, where the dilation is $\log d$ (a packet can traverse at most $\log d$ edges during $\log d$ time steps) and congestion $\log d$. Moreover, a packet has as origin the edge where it arrived at the beginning of current $\log d$ frame, and its destination is its location at the end of the frame. If a packet reaches its destination before the $\log d$ frame finishes, then it is buffered in the queue at its destination until the next $\log d$ frame starts (to be able to treat scheduling problems across frames separately). In the second recursive step, there exists a (non-legitimate) schedule with makespan, say $(1+\beta)\log d$, where the maximum number of packets that can traverse any edge at any time is $\log\log d$ during a time frame of length $\log d$ (Lemma 2). The third recursive step would then divide the subproblems into $(1+\beta)\log d/\log\log d$ subproblems, each of length $\log\log d$, etc. Keep dividing the time frames in this manner until the number of packets in the subproblem becomes constant in d; this happens after $O(\log^* d)$ recursion steps. Thus, the leaves of recursion tree are at depth $O(\log^* d)$ in the recursion tree. At every recursion step, we notice that the makespan of the resulting schedule increases by a constant factor, from which we get a total increase of constant $2^{O(\log^* d)}$ across all recursion levels. Thus, the makespan of the resulting schedule is $d2^{O(\log^* d)}$. Therefore, the schedule at each recursive step can be turned

to a legitimate one by simulating it a constant number of times so that each packet uses exactly at most edge.

To make things a bit more formal, denote as $M(d)$ the makespan of the schedule when the input is of size d. Then we get the following recurrence relation for our the makespan of our algorithm

$$M(d) \leqslant \frac{(1+\alpha)d}{\log d} M(\log d),$$

with $M(0) = 0$ and $M(1) = 1$. If we make the guess $M(d) = d2^{O(\log^* d)}$, then

$$d2^{O(\log^* d)} \leqslant \frac{(1+\alpha)d}{\log d} \log d 2^{O(\log^* \log d)} = (1+\alpha)d2^{O(\log^* d)}$$

holds, because both $2^{O(\log^* d)}$ and $2^{O(\log^* \log d)}$ are constants (pedantically, $\log^* \log d = \log^* d - 1$). Moreover, the increase of the congestion at each recursive step is constant as well, and so we get a constant $2^{O(\log^* d)}$ increase in the original $\log d$ congestion as a result of the new schedule, yielding a $\log d 2^{O(\log^* d)}$ overall congestion of the final schedule.

□

3.2 Applying the Moser-Tardos LLL Algorithm to Construct the Schedule

With the Moser-Tardos algorithm for LLL in hand, we can construct the schedule whose existence we have just shown in reasonable expected number of steps in the probability, p, of our m bad events. The first step in applying the algorithm is to create a set of random variables whose assignment encodes the solution of the problem. A solution in our case is a set of packet delays, each chosen from the set $\{1, \ldots, \alpha d\}$. Thus, we associate with each of our n input packets a delay random variable, D, which assumes values in $\{1, \ldots, \alpha d\}$. Denote the set of all our n random variables as $\mathcal{D}$. Let the set of edge bad events be $\mathcal{E} = \{E_1, \ldots, E_m\}$, where we assume a fixed labeling $(a_1, \ldots a_m)$ on the input network edges. Every bad event E_i is determined by at most $c = d$ random variables, which is given by the congestion of the packet paths. Thus, for every $i \in \{1, \ldots, m\}$, $\texttt{vbl}(E_i)$ is the set of random variables in $\mathcal{D}$ associated with every packet whose given path contains the edge a_i. We have shown that two edges are dependent iff a packet passes through both edges, so the dependency graph of $E_1, \ldots, E_m$ is well defined, and every vertex in this graph cannot have more than d^2 neighbors.

Now that the algorithmic setup is complete, we run the Moser-Tardos algorithm (Algorithm 1) on $\mathcal{D}$ as the input random variables and $\mathcal{E}$ as the set of bad events. The algorithm will terminate with an assignment $(v_D)_{D \in \mathcal{D}}$ of packet delays, where $v_D \in \{1, \ldots, \alpha d\}$. Packet D will wait in its initial queue for v_D time steps deterministically before it starts traversing the network. Thus, constructing the schedule function $\sigma(t)$ is routine: equip every edge with a queue of size $\log(c+d)2^{O(\log^*(c+d))}$, then simulate every time step given the initial packet delays, for at most $(c+d)2^{O(\log^*(c+d))}$ time steps. At every time step t, move the packets that are at the head of edge queues through their next edges in their paths and create the snapshot vector $(J_1, \ldots, J_m)$ accordingly and associate it with t, and advance the location of all other packets waiting in edge queues.

4 Concluding Remarks

We presented the Lovász Local Lemma in both its symmetric and general asymmetric forms. We presented the Algorithmic LLL, due to Moser and Tardos, which finds the points in the sample space over which a finite set of bad events are to be avoided, given that the bad events are locally dependent upon each other, and their probabilities satisfy the requirements of the general asymmetric LLL. We examined the proof of correctness of the algorithmic LLL in full detail, and showed that the algorithm does terminate in a reasonable expected number of steps with respect to bad event probabilites. We then presented an application of LLL to packet routing in networks, and showed that a schedule exists with length that is almost linear in the size of the input, namely the magnitudes of the input network dilation and congestion.

We take the opportunity to show the connection between packet routing as discussed above and job scheduling on parallel machines. The input packets $\mathcal{J}$ become the set of jobs. The edges in the input network become the machines upon which the jobs are to execute, and every machine has its own job queue. The path associated with every job then is the sequence of machines upon which the job is to run until it finishes execution. The maximum number of jobs that need to use a machine simultaneously at any instant is the congestion, c, and the maximum number of machines that need to be used by any job is the dilation, d. Atomicity of packets translates to saying that job execution is *nonpreemptive*: Once a job starts execution upon a machine, it should run to completion on that machine without interruption. The machines are identical in our setting, in that a job's execution requirement is indifferent to the machine upon which it executes, and is constant across all machines. With the objective of minimizing the makespan of the schedule, the problem becomes a variation of the *Job Shop Scheduling problem on m machines with no recirculation and no preemption* (See Pinedo [6], chap. 7). The no recirculation requirement comes from the simple paths assumption imposed on the packet routing problem. As a consequence of the packet routing results, a schedule of makespan $O(c+d)$ does exist for the above-mentioned job shop scheduling problem, where at most a constant number of jobs wait in machine queues, and jobs wait for a constant number of steps in the queues of intermediate machines, after they wait for at most $O(c+d)$ steps at their initial queues.

References

[1] K. B. Athreya and P. E. Ney. *Branching Processes*. Dover Publications, March 19 2004. ISBN 0486434745.

[2] R. Durrett. *Probability Theory and Examples*. Cambridge University Press, Cambridge, August 2010. ISBN 9780521765398.

[3] P. Erdős and L. Lovász. Problems And Results On 3Chromatic Hypergraphs and Some Related Questions. 1975.

[4] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in o(congestion+dilation) steps. *Combinatorica*, 1994.

[5] R. A. Moser and G. Tardos. A constructive proof of the general lovász local lemma. *J. ACM*, 57:11:1–11:15, February 2010. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/1667053.1667060. URL http://doi.acm.org/10.1145/1667053.1667060.

[6] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 3 edition, July 24, 2008.

[7] J. Shearer. On a problem of spencer. *Combinatorica*, 5:241–245, 1985. ISSN 0209-9683. URL http://dx.doi.org/10.1007/BF02579368. 10.1007/BF02579368.

[8] J. Spencer. Asymptotic lower bounds for ramsey functions. *Discrete Mathematics*, 20(0):69 – 76, 1977. ISSN 0012-365X. doi: 10.1016/0012-365X(77)90044-9. URL http://www.sciencedirect.com/science/article/pii/0012365X77900449.

A Proofs of Some Propositions

Here we prove some propositions that we used in the article.

Proposition 2. *If A and B are independent events, then their complements A^c and B^c are independent.*

Proof. If A and B are independent, then

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B). \tag{7}$$

Write A as the union of two disjoint events: $A = (A \cap B) \cup (A \cap B^c)$. Then

$$\mathbb{P}(A) = \mathbb{P}(A \cap B) + \mathbb{P}(A \cap B^c).$$

Substituting $\mathbb{P}(A \cap B)$ for $\mathbb{P}(A) - \mathbb{P}(A \cap B^c)$ in Eqn. (7), we get

$$\mathbb{P}(A) - \mathbb{P}(A \cap B^c) = \mathbb{P}(A)\mathbb{P}(B),$$

so

$$\mathbb{P}(A \cap B^c) = \mathbb{P}(A) - \mathbb{P}(A)\mathbb{P}(B) = \mathbb{P}(A)(1 - \mathbb{P}(B)) = \mathbb{P}(A)\mathbb{P}(B^c). \tag{8}$$

Now write $B^c = (A \cap B^c) \cup (A^c \cap B^c)$. Thus

$$\mathbb{P}(B^c) = \mathbb{P}(A \cap B^c) + \mathbb{P}(A^c \cap B^c).$$

Substituting $\mathbb{P}(A \cap B^c)$ for $\mathbb{P}(B^c) - \mathbb{P}(A^c \cap B^c)$ in Eqn. (8), we get

$$\mathbb{P}(B^c) - \mathbb{P}(A^c \cap B^c) = \mathbb{P}(A)\mathbb{P}(B^c),$$

from which we get

$$\mathbb{P}(A^c \cap B^c) = \mathbb{P}(B^c) - \mathbb{P}(A)\mathbb{P}(B^c) = \mathbb{P}(B^c)(1 - \mathbb{P}(A)) = \mathbb{P}(A^c)\mathbb{P}(B^c).$$

□

Next we show the result for n mutually independent events.

Proposition 3. *Let $A_1, \ldots, A_n$ be n jointly (mutually) independent events over some probability space. Then their complements $A_1^c, \ldots, A_n^c$ are mutually independent.*

Proof. We show that if $A_1, \ldots, A_n$ are mutually independent, then $A_1^c, A_2, \ldots, A_n$ are mutually independent. Let $A_{i_1}, \ldots, A_{i_r}$ be an r subset of $A_1, \ldots, A_n$. If A_1^c is not any of $A_{i_1}, \ldots, A_{i_r}$, then

$$\mathbb{P}\left(\bigcap_{j=1}^{r} A_{i_j}\right) = \prod_{j=1}^{r} \mathbb{P}(A_{i_j}).$$

If A_1^c is one of $A_{i_1}, \ldots, A_{i_r}$, then wlog, let $A_1^c = A_{i_1}$. Then

$$\begin{aligned}
\mathbb{P}\left(\bigcap_{j=1}^{r} A_{i_j}\right) &= \mathbb{P}\left(A_1^c \cap \bigcap_{j=2}^{r} A_{i_j}\right) = \mathbb{P}\left(A_1^c \cap \bigcap_{j=2}^{r} A_{i_j}\right) \mathbb{P}\left((\Omega \setminus A_1) \cap \bigcap_{j=2}^{r} A_{i_j}\right) \\
&= \mathbb{P}\left(\left(\bigcap_{j=2}^{r} A_{i_j}\right) \setminus \left(A_1 \cap \bigcap_{j=2}^{r} A_{i_j}\right)\right) = \prod_{j=2}^{r} \mathbb{P}(A_{i_j}) - \mathbb{P}(A_1) \prod_{j=2}^{r} \mathbb{P}(A_{i_j}) \\
&= (1 - \mathbb{P}(A_1)) \prod_{j=2}^{r} \mathbb{P}(A_{i_j}) = \mathbb{P}(A_1^c) \prod_{j=2}^{r} \mathbb{P}(A_{i_j}).
\end{aligned}$$

Thus if $A_1, \ldots, A_n$ are mutually independent, then $A_1^c, A_2, \ldots, A_n$ are mutually independent. Repeating this argument on the last result, one shows that $A_1^c, A_2, \ldots, A_n$ are mutually independent, then $A_1^c, A_2^c, A_3, \ldots, A_n$ are mutually independent. By induction, the desired result will follows when one shows that if $A_1^c, \ldots, A_{n-1}^c, A_n$ are mutually independent, then so are $A_1^c, \ldots, A_n^c$. □

B The Iterated Logarithm Function

The iterated logarithm function $\log^* : \mathbb{R}_+ \to \mathbb{Z}_+$ counts the number of times log (in our case, the binary logarithm $\log_2$) has to be applied to its argument before the result of the successive applications of the logarithm becomes less than one. Formally

$$\log^* n = \begin{cases} 1 + \log^*(\log n) & \text{if } n \leqslant 1, \\ 0 & \text{if } n > 1. \end{cases}$$

Consider, for example, an algorithmic problem of input size n. Suppose that a divide-and-conquer algorithm that solves the problem operates as follows: at each recursive step, the algorithm divides the problem into sub-problems whose size is logarithmic in the size of the "bigger" subproblem in the previous recursive step. When the size of the subproblem becomes constant in n, it solves it. Finally, the algorithm combines the solution of the subproblems to produce a solution for the larger subproblems. At the first recursive step, the algorithm divides n into $n/\log n$ subproblems, each of which has size $\log n$. Then, for each of those $\log n$-sized subproblems, the algorithm divides the input into $\log n/\log\log n$ subproblems, each of size $\log\log n$, etc. The division process continues until the size of the input of each subproblem becomes constant in n; this happens when the size of the input is $\log^* n$. Therefore, the length of the recursion tree of this algorithm is $\log^* n$.

To appreciate how slowly $\log^* n$ grows with respect to n, consider the tower function

$$f(n) = 2^{2^{2^{2^{2^n}}}} .$$

For $n = 3$, $f(3) = 2^{2^{2^{2^{2^3}}}} = 2^{2^{2^{2^8}}} = 2^{2^{2^{2^{256}}}} = 2^{2^{1.15792089237316195423570985008687907853269984665640564039457584007913129639936 \times 10^{77}}} = \cdots$ (our calculator could not handle more !) However, consider computing $\log^*(f(3))$: $\log(f(3)) = \log(2^{2^{2^{2^{2^3}}}}) = 2^{2^{2^{2^3}}}$, $\log\log(f(3)) = \log(2^{2^{2^{2^3}}}) = 2^{2^{2^3}}$, $\log\log\log(f(3)) = \log(2^{2^{2^3}}) = 2^{2^3}$, $\log\log\log\log(f(3)) = \log(2^{2^3}) = 2^3$, $\log\log\log\log\log(f(3)) = \log(2^3) = 3$, and finally, $\log\log\log\log\log\log(f(3)) = \log(3) < 1$. Thus, $\log^*(f(3)) = 6$ only ! This is almost constant in $f(3)$. In fact, it has been noticed that $\log^*(n)$ is rarely beyond 5 for any problem of practical input size to be solved by a machine (which happens when the input, n, is as large in magnitude as 2^{65536}; this number is greater than the number of atoms in the observable universe !)