A constructive algorithm for the Lovász Local Lemma on permutations

David G. Harris* Aravind Srinivasan†

Abstract

While there has been significant progress on algorithmic aspects of the Lovász Local Lemma (LLL) in recent years, a noteworthy exception is when the LLL is used in the context of random permutations: the "lopsided" version of the LLL is usually at play here, and we do not yet have subexponential-time algorithms. We resolve this by developing a randomized polynomial-time algorithm for such applications. A noteworthy application is for Latin Transversals: the best-known general result here (Bissacot et al., improving on Erdős and Spencer), states that any $n \times n$ matrix in which each entry appears at most $(27/256)n$ times, has a Latin transversal. We present the first polynomial-time algorithm to construct such a transversal. Our approach also yields RNC algorithms: for Latin transversals, as well as the first efficient ones for the strong chromatic number and (special cases of) acyclic edge-coloring.

1 Introduction

Recent years have seen substantial progress on developing algorithmic versions of the Lovász Local Lemma (LLL) and some of its generalizations, starting with the breakthrough work of Moser & Tardos [31]: see, e.g., [20, 21, 25, 34]. However, one major relative of the LLL that has eluded constructive versions, is the "lopsided" version of the LLL (with the single exception of the CNF-SAT problem [31]). A natural setting for the lopsided LLL is where we have one or many random permutations [16, 26, 29]. This approach has been used for Latin transversals [9, 16, 39], hypergraph packing [27], certain types of graph coloring [10], and in proving the existence of certain error-correcting codes [24]. However, current techniques do not give constructive versions in this context. We develop a randomized polynomial-time algorithm to construct such permutation(s) whose existence is guaranteed by the lopsided LLL, leading to several algorithmic applications in combinatorics.

1.1 The Lopsided Local Lemma and Random Permutations

Suppose we want to select permutations $\pi_1, \dots, \pi_N$, where each π_k is a permutation on the set $[n_k] = \{1, \dots, n_k\}$. In addition we have a set $\mathcal{B}$ of "bad events." We want to select permutations π such that no bad event is true. The *lopsided version* of the Lovász Local Lemma (LLL) can be used to prove that such permutations exist, under suitable conditions.

We suppose that that the set of bad events $\mathcal{B}$ consists of *atomic bad-events*. Each $B \in \mathcal{B}$ is a set of tuples $B = \{(k_1, x_1, y_1), \dots, (k_r, x_r, y_r)\}$; it is true iff we have $(\pi_{k_1}(x_1) = y_1) \wedge \dots \wedge (\pi_{k_r}(x_r) = y_r)$. (Complex bad-events can usually be decomposed into atomic bad-events, so this does not lose much generality.)

To apply the Lopsided Local Lemma in this setting, we need to define a *dependency graph* with respect to these bad-events. We say two bad events B, B' are *connected* if they overlap in one slice of the domain or range of a permutation; namely, that there are some k, x, y_1, y_2 with $(k, x, y_1) \in B, (k, x, y_2) \in B'$ *or* there are some k, x_1, x_2, y with $(k, x_1, y) \in B, (k, x_2, y) \in B'$. We write this $B \sim B'$; note that $B \sim B$. The following notation will be useful: for pairs $(x_1, y_1), (x_2, y_2)$, we write $(x_1, y_1) \sim (x_2, y_2)$ if $x_1 = x_2$ *or* $y_1 = y_2$ (or both). Another way to write $B \sim B'$ is that "there are $(k, x, y) \in B$, $(k, x', y') \in B'$ with $(x, y) \sim (x', y')$".

We will use the following notation at various points: we write $(k, x, *)$ to mean any (or all) triples of the form (k, x, y), and similarly for $(k, *, y)$, or $(x, *)$ etc. Another way to write the condition $B \sim B'$ is that there are $(k, x, *) \in B, (k, x, *) \in B'$ or $(k, *, y) \in B, (k, *, y) \in B'$.

Now suppose we select each π_k uniformly at random and independently. This defines a probability space, to which we can apply the lopsided LLL. One can show that the probability of avoiding a bad event B can only be *increased* by avoiding other bad events $B' \not\sim B$ [27]. Thus, in the language of the lopsided LLL, the relation $\sim$ defines a *negative-dependence* graph among the bad-events. (See [26, 27, 29] for a study of the connection between negative dependence, random injections/permutations, and the lopsided LLL.) Hence, the standard lopsided-LLL criterion is as follows:

THEOREM 1.1. ([27]) *Suppose that there is some assignment $\mu : \mathcal{B} \to [0, \infty)$ such that for all bad-events*

*Department of Applied Mathematics, University of Maryland, College Park, MD 20742. Research supported in part by NSF Award CNS-1010789. Email: davidgharris29@hotmail.com

†Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Research supported in part by NSF Award CNS-1010789. Email: srin@cs.umd.edu.

$B \in \mathcal{B}$ we have

$$\mu(B) \geq P(B) \prod_{B' \sim B} (1 + \mu(B')).$$

Then the random process has a positive probability of selecting permutations $\pi_1, \pi_2, \ldots, \pi_N$ which avoid all the bad-events.[1]

The positive probability here is however typically exponentially small, as is standard for the LLL. As mentioned above, a variety of papers have used this framework for proving the existence of various combinatorial structures. Unfortunately, the algorithms for the LLL, such as Moser-Tardos resampling [31], do not apply in this setting. The problem is that such algorithms have a more restrictive notion of when two bad-events are dependent; namely, that they share variables. (The Moser-Tardos algorithm allows for a restricted type of dependence called *lopsidependence*: two bad-events which share a variable but always *agree* on that value, are counted as independent. This is not strong enough to generate permutations.) So we do not have an algorithm to generate such permutations, we can merely show that they exist.

We develop an algorithmic analog of the LLL for permutations. The necessary conditions for our Swapping Algorithm are the same as for the probabilistic LLL (Theorem 1.1); however, we will construct such permutations in randomized polynomial (typically linear or near-linear) time. Our setting is far more complex than in similar contexts such as those of [31, 21, 34], and requires many intermediate results first. The main complication is that when we encounter a bad event involving "$\pi_k(x) = y$", and we perform our algorithm's random swap associated with it, we could potentially be changing any entry of π_k. In contrast, when we resample a variable in [31, 21, 34], all the changes are confined to that variable. There is a further technical issue, which is that the current witness-tree-based algorithmic versions of the LLL, such as [31, 21], identify, for each bad-event B in the witness-tree τ, some necessary event occurring with probability at most $P(B)$. This is not the proof we employ here; there are significant additional terms ("$(n_k - A_k^0)!/n!$" – see the proof of Lemma 3.1) that are gradually "discharged" over time. We also develop RNC versions of our algorithms. Going from serial to parallel is fairly direct in [31]; our main bottleneck here is that when we resample an "independent" set of bad events, they could still influence each other.

[1] This condition about the existence of such a μ, is equivalent to the more-familiar LLL formulation "there exists $x : \mathcal{B} \to [0, 1)$ such that for all $B \in \mathcal{B}$, $P(B) \leq x(B) \prod_{B' \sim B : B' \neq B} (1 - x(B'))$": just set $\mu(B) = x(B)/(1 - x(B))$.

1.2 Applications We present algorithmic applications for three classical combinatorial problems: Latin transversals, the strong chromatic number, and acyclic edge-coloring. In addition to the improved bounds, we wish to highlight two features here. First, our algorithmic approach goes beyond Theorem 1.1: as we will see shortly, one of our (asymptotically-optimal) algorithmic results on Latin transversals, could not even have been shown nonconstructively using the lopsided LLL prior to this work. Second, the acyclic-edge-coloring application is striking in that its statement does not involve permutations; indeed, all existing Local Lemma-based results for acyclic edge-coloring use the standard ("asymmetric") LLL, not the lopsided version. We reduce the problem to one involving (random) injections, to which our framework is applicable. Since coloring problems naturally come with injection (distinctness) requirements, we hope this method will be applicable to other coloring problems as well.

The study of Latin squares and the closely-related Latin transversals is a classical area of combinatorics, going back to Euler and earlier [12]. Given an $m \times n$ matrix A with $m \leq n$, a *transversal* of A is a choice of m elements from A, one from each row and at most one from any column. Perhaps the major open problem here is: given an integer s, under what conditions will A have an *s-transversal*: a transversal in which no value appears more than s times [9, 15, 16, 37, 38, 39]? The usual type of sufficient condition sought here is an upper bound Δ on the number of occurrences of any given value in A. That is, we ask: what is the maximum $\Delta = \Delta(s; m, n)$ such that any $m \times n$ matrix A in which each value appears at most Δ times, is guaranteed to have an s-transversal? The case $s = 1$ is perhaps most studied, and 1-transversals are also called *Latin transversals*. The case $m = n$ is also commonly studied (and includes Latin squares as a special case), and we will also focus on these. It is well-known that $L(1; n, n) \leq n - 1$ [38]. In perhaps the first application of the lopsided LLL to random permutations, Erdős & Spencer essentially proved a result very similar to Theorem 1.1, and used it to show that $L(1; n, n) \geq n/(4e)$ [16]. (Their paper shows that $L(1; n, n) \geq n/16$; the $n/(4e)$ lower-bound follows easily from their technique.) To our knowledge, this is the first $\Omega(n)$ lower-bound on $L(1; n, n)$. Alon asked if there is a constructive version of this result [2]. Building on [16] and using the connections to the LLL from [35, 36], Bissacot *et al.* showed nonconstructively that $L(1; n, n) \geq (27/256)n$ [9]. Our result makes this constructive.

The lopsided LLL has also been used to study the case $s > 1$ [39]. Here, we prove a result that is asymptotically optimal for large s, except for the

lower-order $O(\sqrt{s})$ term: we show (algorithmically) that $L(s;n,n) \geq (s - O(\sqrt{s})) \cdot n$. An interesting fact is that this was not known even nonconstructively before: Theorem 1.1 roughly gives $L(s;n,n) \geq (s/e) \cdot n$. We also give faster serial and perhaps the first RNC algorithms with good bounds, for the strong chromatic number. Strong coloring is quite well-studied [3, 8, 18, 22, 23], and is in turn useful in *covering* a matrix with Latin transversals [6]. Finally, we develop the first RNC algorithms for (special cases of) acyclic edge-coloring, a well-studied problem [4, 7, 19, 17, 30, 32, 33]. As mentioned above, this is a problem that does not appear to involve permutations; we hope our methods based on (random) injections and the permutation LLL, will be useful for other coloring problems also.

2 The Swapping Algorithm

We will analyze the following *Swapping Algorithm* algorithm to find a satisfactory $\pi_1, \ldots, \pi_N$:

1. Generate the permutations $\pi_1, \ldots, \pi_N$ uniformly at random and independently.
2. While there is some true bad-event:
    3. Choose some true bad-event $B \in \mathcal{B}$ arbitrarily. For each permutation that is involved in B, we perform a *swapping* of all the relevant entries. (We will describe the swapping subroutine "Swap" shortly.) We refer to this step as a *resampling* of the bad-event B.

       Each permutation involved in B is swapped independently, but if B involves multiple entries from a single permutation, then all such entries are swapped *simultaneously.* For example, if B consisted of triples $(k_1, x_1, y_1), (k_2, x_2, y_2), (k_2, x_3, y_3)$, then we would perform Swap$(\pi_1; x_1)$ and Swap$(\pi_2; x_2, x_3)$, where the "Swap" procedure is given next.

The swapping subroutine Swap$(\pi; x_1, \ldots, x_r)$ for a permutation $\pi : [t] \to [t]$ as follows:

Repeat the following for $i = 1, \ldots, r$:

- Select x'_i uniformly at random among $[t] - \{x_1, \ldots, x_{i-1}\}$.
- Swap entries x_i and x'_i of π.

Note that at every stage of this algorithm all the π_k are permutations, and if this algorithm terminates, then the π_k must avoid all the bad-events. So our task will be to show that the algorithm terminates in polynomial time. We measure time in terms of a single iteration of the main loop of the Swapping Algorithm: each time we run one such iteration, we increment the time by one. We will use the notation π_k^T to denote the value of permutation π_k after time T. The initial sampling of the permutation (after Step (1)) generates π_k^0.

The swapping subroutine seems strange; it would appear more natural to allow x'_i to be uniformly selected among $[t]$. However, the swapping subroutine is nothing more than than the Fisher-Yates Shuffle for generating uniformly-random permutations. If we allowed x'_i to be chosen from $[t]$ then the resulting permutation would be biased. The goal is to change π_k in the minimal way to ensure that $\pi_k(x_1), \ldots, \pi_k(x_r), \pi_k^{-1}(y_1), \ldots, \pi_k^{-1}(y_r)$ are adequately randomized.

There are alternative methods for generating random permutations, and many of these have equivalent behavior to our Swapping subroutine. We will sometimes use such equivalences without proof. One class of algorithms that has a very different behavior is the commonly used method to generate random reals $r_i \in [0, 1]$, and then form the permutation by sorting these reals. When encountering a bad-event, one would resample the affected reals r_i. In our setting, where the bad-events are defined in terms of specific values of the permutation, this is not a good swapping method because a single swap can drastically change the permutation. When bad-events are defined in terms of the relative *rankings* of the permutation (e.g. a bad event is $\pi(x_1) < \pi(x_2) < \pi(x_3)$), then this is a better method and can be analyzed in the framework of the ordinary Moser-Tardos algorithm.

3 Witness trees and witness dags

To analyze the Swapping Algorithm, following the Moser-Tardos approach [31], we introduce the concept of an execution log and a witness tree. The execution log consists of listing every resampled bad-event, in the order that they are resampled. We form a witness tree to justify a certain resampling E, for example the last resampling. We start with the event E, and create a single node in our tree labeled by this event. We move backward in time; for each bad-event B we encounter, we add it to the witness tree if $B \sim B'$ for some event B' already in the tree: we choose such a B' that has the maximum depth in the current tree (breaking ties arbitrarily), and make B a child of this B' (there could be many nodes labeled B'). If $B \not\sim B'$ for all B' in the current tree, we ignore this B and keep moving backward in time. To make this discussion simpler we say that the root of the tree is at the "top" and the deep layers of the tree are at the "bottom". The top of the tree corresponds to later events, the bottom of the tree to the earliest events.

For the remainder of this section, the dependence on the "justified" bad-event E at the root of the tree will be understood; we will omit it from the notation.

We will use the term "witness tree" in two closely-related senses in the following proof. First, when we run the Swapping Algorithm, we produce a witness tree $\hat{\tau}$; this is a random variable. Second, we might want to fix some labeled tree τ, and discuss hypothetically under what conditions it could be produced or what properties it has; in this sense, τ is a specific object. We will always use the notation $\hat{\tau}$ to denote the specific witness tree which was produced by running the Swapping Algorithm.

The critical lemma that allows us to analyze the behavior of this algorithm is:

LEMMA 3.1. *Let τ be a witness tree, with nodes labeled $B_1, \ldots, B_s$. For any event E, the probability that τ was produced as the witness tree corresponding to event E, is at most*

$$P(\hat{\tau} = \tau) \leq P(B_1) \cdots P(B_s)$$

We define the probability of a bad-event $P(B)$ as follows: if the event contains $r_1, \ldots, r_N$ elements from each of the permutations $1, \ldots, N$, then we have

$$P(B) = \frac{(n_1 - r_1)!}{n_1!} \cdots \frac{(n_N - r_N)!}{n_N!}$$

This is simply the probability that B would occur, if all the permutations had been chosen uniformly at random initially.

This lemma is superficially similar to the corresponding lemma in Moser-Tardos [31]. However, the proof will be far more complex, and we will require many intermediate results first. The main complication is that when we encounter a bad-event involving $\pi_k(x) = y$, and we perform the random swap associated with it, then we could potentially be changing any entry of π_k. By contrast, in the usual Moser-Tardos algorithm, when we resample a variable, all the changes are confined to that variable. However, as we will see, the witness tree will leave us with enough clues about which swap was actually performed that we will be able to narrow down the possible impact of the swap.

The analysis in the next sections can be very complicated. We have two recommendations to make these proofs easier. First, the basic idea behind how to form and analyze these trees comes from [31]; the reader should consult that paper for results and examples which we omit here. Second, one can get most of the intuition behind these proofs by considering the situation in which there is a single permutation, and the bad-events all involve just a single element; that is, every bad-event has the form $\pi(x_i) = y_i$. In this case, the witness dags (defined later) are more or less equivalent to the witness tree. (The main point of the witness dag concept is, in effect, to reduce bad-events to their individual elements.) When reading the following proofs, it is a good idea to keep this special case in mind. In several places, we will discuss how certain results simplify in that setting.

The following proposition is the main reason the witness tree encodes sufficient information about the sequence of swaps:

PROPOSITION 3.1. *Suppose that at some time t we have $\pi_k^t(x) \neq y$, and at some later time $t' > t$ we have $\pi_k^{t'}(x) = y$. Then there must have occurred at some intermediate time t'' some bad-event including $(k, x, *)$ or $(k, *, y)$.*

Proof. Let $t'' \in [t, t' - 1]'$ denote the earliest time at which we had $\pi^{t''+1}(x) = y$; this must be due to some swaps $(k, x_1, y_1), \ldots, (k, x_r, y_r)$ at time t''. Suppose that the swap which first caused $\pi(x) = y$ was at swapping x_i of permutation k, which at that time had $\pi_k(x_i) = y_i'$, with some x''.

After this swap, we have $\pi_k(x_i) = y''$ and $\pi_k(x'') = y_i'$. Evidently $x'' = x$ or $x_i = x$. In the first case, the bad event at time t'' included $(k, x, *)$ as desired and we are done.

So suppose $x'' = x$ and $y_i' = y$. So at the time of the swap, we had $\pi_k(x_i) = y$. The only earlier swaps in this resampling were with $x_1, \ldots, x_{i-1}$; so we must have had either $\pi_k^{t''}(x_j) = y$ for some $j \leq i$. But, at the beginning of this swap, we had $\pi_k^{t''}(x_j) = y_j$ for all $j = 1, \ldots, r$. This implies that $y = y_j$ for some $j = 1, \ldots, r$; this in turn implies that the bad-event at time t'' did in fact involve $(k, *, y)$ as desired.

In proving Lemma 3.1, we will *not* need to analyze the interactions between the separate permutations, but rather we will be able to handle each permutation in a completely independent way. For a permutation π_k, we define the *witness dag for permutation π_k*; this is a relative of the witness tree, but which only includes the information for a single permutation at a time.

DEFINITION 3.1. (WITNESS DAGS) *For a permutation π_k, a* witness dag for π_k *is defined to be a directed acyclic simple graph, whose nodes are labeled with pairs of the form (x, y). If a node v is labeled by (x, y), we write $v \approx (x, y)$. This graph must in addition satisfy the following properties:*

1. *If any pair of nodes overlaps in a coordinate, that is, we have $v \approx (x, y) \sim (x', y') \approx v'$, then nodes*

v, v' must be comparable (that is, either there is a path from v to v' or vice-versa).

2. *Every node of G has in-degree at most two and out-degree at most two.*

We also may label the nodes with some auxiliary information, for example we will record that the nodes of a witness dag correspond to bad-events or nodes in a witness tree τ.

We will use the same terminology as for witness trees: vertices on the "bottom" are close to the source nodes of G (appearing earliest in time), and vertices on the "top" are close to the sink nodes of G (appear latest in time).

The witness dags that we will be interested in are derived from witness trees in the following manner.

Definition 3.2. (Projection of a witness tree) *For a witness tree τ, we define the* projection of τ onto permutation π_k *which we denote $Proj_k(\tau)$, as follows.*

Suppose we have a node $v \in \tau$ which is labeled by some bad-event $B = (k_1, x_1, y_1), \ldots, (k_r, x_r, y_r)$. For each i with $k_i = k$, we create a corresponding node $v'_i \approx (x_i, y_i)$ in the graph $Proj_k(\tau)$. We also include some auxiliary information indicating that these nodes came from bad event B, and in particular that all such nodes are part of the same bad-event.

We add edges to $Proj_k(\tau)$ as follows. For each node $v' \in Proj_k(\tau)$, labeled by (x, y) and corresponding to $v \in \tau$, we examine all occurrences of nodes labelled $(k, x, *)$. All such occurrences are comparable; we find the node u' corresponding to $u \in \tau$ which is closest to the source. In other words, we find the occurrence u' which appears "next-highest" in the dag. We create an edge from v' to u'. Similarly, we find the next-highest occurrence $w \in \tau$ of a bad-event labeled by $(k, *, y)$; we create an edge from v' to w'.

Note that u, w must appear strictly *higher in τ, because of the way new bad-events are added to witness trees. This implies that $Proj_k(\tau)$ is acyclic. Also, note that it is possible that $u = w$; in this case we only add a single edge to $Proj_k(\tau)$.*

Expository Remark: *In the special case when each bad-event contains a single element, the witness dag is a "flattening" of the tree structure. Each node in the tree corresponds to a node in the witness dag, and each node in the witness dag points to the next highest occurrence of the domain and range variables.*

Basically, the projection of τ onto k tells us all of the swaps of π_k that occur. It also gives us some of the temporal information about these swaps that would have been available from τ. If there is a path from v to v' in $\mathrm{Proj}_k(\tau)$, then we know that the swap corresponding to v must come before the swap corresponding to v'. It is possible that there are a pair of nodes in $\mathrm{Proj}_k(\tau)$ which are incomparable, yet in τ there was enough information to deduce which event came first (because the nodes would have been connected through some other permutation). So $\mathrm{Proj}_k(\tau)$ does discard some information from τ, but it turns out that we will not need this information.

To prove Lemma 3.1, we will prove (almost) the following claim: Let G be a witness dag for permutation π_k; suppose the nodes of G are labeled with bad-events $B_1, \ldots, B_s$. Then the probability that $G = \mathrm{Proj}_k(\hat{\tau})$, is at most

$$P(G = \mathrm{Proj}_k(\hat{\tau})) \leq P_k(B_1) \ldots P_k(B_s)$$

where, for a bad-event B we define $P_k(B)$ in a similar manner to $P(B)$; namely that if the bad-event B contains r_k elements from permutation k, then we have $P_k(B) = \frac{(n_k - r_k)!}{n_k!}$.

Unfortunately, proving this rigorously runs into some technical complications. These can be resolved, but it is cumbersome to do so and we will not directly need this result as stated. However, this should definitely be thought of as the *informal* justification for the analysis in Section 4.

4 The conditions on a permutation π_{k^*} over time

For the purposes of Section 4, we will fix a permutation π_{k^*}. The dependence on $k = k^*$ will be hidden henceforth; we will discuss simply $\pi, Proj(\tau)$, and so forth. We will also hide the dependence on E, which is the seed event for building the witness tree.

In this section, we will describe conditions that π^t must satisfy at various times t during the execution of the Swapping Algorithm. This analysis can be divided into three phases.

1. We define the *future-subgraph* at time t, denoted G_t. This is a kind of graph which encodes necessary conditions on π^t. We define and describe some structural properties of these graphs.

2. We analyze how a future-subgraph G_t imposes conditions on the corresponding permutation π^t, and how these conditions change over time.

3. We compute the probability that the swapping satisfies these conditions.

We will prove (1) and (2) in Section 4. In Section 5 we will put this together to prove (3) for all the permutations.

4.1 The future-subgraph Suppose we have fixed a target graph G, which could hypothetically have been produced as the projection of $\hat{\tau}$ onto π_{k^*}. We begin the execution of the Swapping Algorithm and see if, so far, the dag G appears to have a possibility of appearing as the projection of the witness tree. This is a kind of dynamic process, in which we are monitoring whether it is still possible to have $G = \text{Proj}(\hat{\tau})$ and, if so, what conditions this would entail.

We define the *future-subgraph* of G at time t, denoted G_t, as follows.

DEFINITION 4.1. (THE FUTURE-SUBGRAPH) *Initially $G_0 = G$. We define the future-subgraphs G_t for $t > 0$ inductively. When we run the Swapping Algorithm, as we encounter a bad-event $(k_1, x_1, y_1), \ldots, (k_r, x_r, y_r)$, we update G_t as follows:*

1. *Suppose that $k_i = k^*$, and G_t contains a source node labeled (x_i, y_i). We remove that source node from G_t.*
2. *Suppose that $k_i = k^*$, and G_t has a source labeled (x_i, y'') where $y'' \neq y_i$ or (x'', y_i) where $x'' \neq x_i$. Then, as will be shown in shown in Proposition 4.1, we can immediately conclude G is impossible; we set $G_{t+1} = \perp$, and we can abort the execution of the Swapping Algorithm.*

We let G_{t+1} denote the graph after applying these updates.

The following Proposition tells us that G_t correctly describes the future swaps that must occur:

PROPOSITION 4.1. *For any time $t \geq 0$, let $\hat{\tau}_{\geq t}$ denote the witness tree built for the event E, but only using the execution log from time t onwards. Then if $Proj(\hat{\tau}) = G$ we also have $Proj(\hat{\tau}_{\geq t}) = G_t$.*

Note that if $G_t = \perp$, the latter condition is obviously impossible; in this case, we are asserting that whenever $G_t = \perp$, it is impossible to have $Proj(\hat{\tau}) = G$.

Proof. We prove this by induction on t. When $t = 0$, this is obviously true as $\hat{\tau}_{\geq 0} = \hat{\tau}$ and $G_0 = G$.

Suppose we have $\text{Proj}(\hat{\tau}) = G$; at time t we encounter a bad-event $B = (k_1, x_1, y_1), \ldots, (k_r, x_r, y_r)$. By inductive hypothesis, $\text{Proj}(\hat{\tau}_{\geq t}) = G_t$.

Suppose first that $\hat{\tau}_{\geq t+1}$ does not contain any bad-events $B' \sim B$. Then, by our rule for building the witness tree, we have $\hat{\tau}_{\geq t} = \hat{\tau}_{\geq t+1}$. Hence we have $G_t = \text{Proj}(\hat{\tau}_{\geq t+1})$. When we project this graph onto permutation k, there cannot be any source node labeled (k, x, y) with $(x, y) \sim (x_i, y_i)$ as such node would be labeled with $B' \sim B$. Hence, according to our rules for updating G_t, we have $G_{t+1} = G_t$. So in this case we have $\hat{\tau}_{\geq t} = \hat{\tau}_{\geq t+1}$ and $G_t = G_{t+1}$ and $\text{Proj}(\hat{\tau}_{\geq t}) = G_t$; it follows that $\text{Proj}(\hat{\tau}_{\geq t+1}) = G_{t+1}$ as desired.

Next, suppose $\hat{\tau}_{\geq t+1}$ does contain $B' \sim B$. Then bad-event B will be added to $\hat{\tau}_{\geq t}$, placed below any such B'. When we project $\hat{\tau}_{\geq t}$, then for each i with $k_i = k^*$ we add a node (x_i, y_i) to $\text{Proj}(\hat{\tau}_{\geq t})$. Each such node is necessarily a source node; if such a node (x_i, y_i) had a predecessor $(x'', y'') \sim (x_i, y_i)$, then the node (x'', y'') would correspond to an event $B'' \sim B$ placed below B. Hence we see that $\text{Proj}(\hat{\tau}_{\geq t})$ is obtained from $\text{Proj}(\hat{\tau}_{\geq t})$ by adding source nodes (x_i, y_i) for each $(k^*, x_i, y_i) \in B$.

So $\text{Proj}(\hat{\tau}_{\geq t}) = \text{Proj}(\hat{\tau}_{\geq t+1})$ plus the addition of source nodes for each (k^*, x_i, y_i). By inductive hypothesis, $G_t = \text{Proj}(\hat{\tau}_{\geq t})$, so that $G_t = \text{Proj}(\hat{\tau}_{\geq t+1})$ plus source nodes for each (k^*, x_i, y_i). Now our rule for updating G_{t+1} from G_t is to remove all such source nodes, so it is clear that $G_{t+1} = \text{Proj}(\hat{\tau}_{\geq t+1})$, as desired.

Note that in this proof, we assumed that $\text{Proj}(\hat{\tau}) = G$, and we never encountered the case in which $G_{t+1} = \perp$. This confirms our claim that whenever $G_{t+1} = \perp$ it is impossible to have $\text{Proj}(\hat{\tau}) = G$.

By Proposition 4.1, the witness dag G and the future-subgraphs G_t have a similar shape; they are all produced by projecting witness trees of (possibly truncated) execution logs. Note that if $G = \text{Proj}(\tau)$ for some tree τ, then for any bad-event $B \in \tau$, either B is not represented in G, or all the pairs of the form $(k^*, x, y) \in B$ are represented in G and are incomparable there.

The following structural decomposition of a witness dag G will be critical.

DEFINITION 4.2. (ALTERNATING PATHS) *Given a witness dag G, we define an* alternating path *in G to be a simple path which alternately proceeds forward and backward along the directed edges of G. For a vertex $v \in G$, the* forward (respectively backward) path *of v in G, is the maximal alternating path which includes v and all the forward (respectively backward) edges emanating from v. Because G has in-degree and out-degree at most two, every vertex v has a unique forward and backward path (up to reflection); this justifies our reference to "the" forward and backward path. These paths may be even-length cycles.*

Note that if v is a source node, then its backward path contains just v itself. This is an important type of alternating path which should always be taken into account in our definitions.

One type of alternating path, which is referred to as the *W-configuration*, will play a particularly important role.

DEFINITION 4.3. (THE W-CONFIGURATION) *Suppose $v \approx (x, y)$ has in-degree at most one, and the backward path contains an* even *number of edges, terminating at vertex $v' \approx (x', y')$. We refer to this alternating path as a* W-configuration. *(See Figure 1.)*

Any W-configuration can be written (in one of its two orientations) as a path of vertices labeled $(x_0, y_1), (x_1, y_1), (x_2, y_1), \ldots, (x_s, y_s), (x_s, y_{s+1})$; here the vertices $(x_1, y_1), \ldots, (x_s, y_s)$ are at the "base" of the W-configuration. Note here that we have written the path so that the x-coordinate changes, then the y-coordinate, then x, and so on. When written this way, we refer to (x_0, y_{s+1}) as the endpoints *of the W-configuration.*

If $v \approx (x, y)$ is a source node, then it defines a W-configuration with endpoints (x, y). This should not be considered a triviality or degeneracy, rather it will be the most important type of W-configuration.

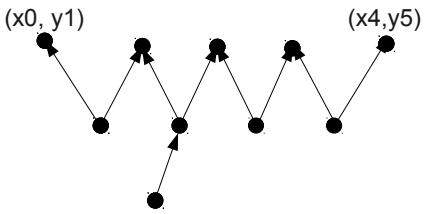

Figure 1: A W-configuration of length 9, with endpoints (x_0, y_5).

4.2 The conditions on $\pi^t_{k^*}$ encoded by G_t At any given time t, the current future-subgraph G_t gives certain necessary conditions on π. Proposition 4.2 describes a certain set of conditions that plays a key role in the analysis.

PROPOSITION 4.2. *Suppose we have a witness dag G for permutation π. Let π^t denote the value of permutation π at time t and G_t be the future-subgraph at time t.*

A necessary condition to have $G = Proj(\hat{\tau})$ is the following: For every W-configuration in G_t with endpoints (x_0, y_{s+1}), we must have $\pi^t(x_0) = y_{s+1}$.

For example, if $v \approx (x, y)$ is a source node of G_t, then $\pi^t(x) = y$.

Proof. We prove this by induction on s. The base case is $s = 0$; in this case we have a source node (x, y). Suppose $\pi^t(x) \neq y$. In order for the witness tree to contain some bad-event containing (k^*, x, y), we must at some point $t' > t$ have $\pi^{t'}(x) = y$; let t' be the minimal such time. By Proposition 3.1, we must encounter a bad-event containing $(k^*, x, *)$ or $(k^*, *, y)$ at some intervening time $t'' < t'$. If this bad-event contains (k^*, x, y) then necessarily $\pi^{t''}(x) = y$ contradicting minimality of t'. So there is a bad-event $(k^*, x, \neq y)$ or $(k^*, \neq x, y)$ earlier than the earliest occurrence of $\pi(x) = y$. This event $(k^*, x, \neq y)$ or $(k^*, \neq x, y)$ projects to a source node $(x, \neq y)$ or $(\neq x, y)$ in G_t. But then (x, y) cannot also be a source node of G_t.

We now prove the induction step. Suppose we have a W-configuration with base $(x_1, y_1), \ldots, (x_s, y_s)$. At some future time $t' \geq t$ we must encounter a bad-event involving some subset of the source nodes, say the bad event includes $(x_{i_1}, y_{i_1}), \ldots, (x_{i_r}, y_{i_r})$ for $1 \leq r \leq s$. As these were necessarily source nodes, we had $\pi^{t'}(x_{i_1}) = y_{i_1}, \ldots, \pi^{t'}(x_{i_r}) = y_{i_r}$ for $l = 1, \ldots, r$. After the swaps, the updated G_{t+1} has $r + 1$ new W-configurations. By inductive hypothesis, the updated permutation $\pi^{t'+1}$ must then satisfy $\pi^{t'+1}(x_0) = y_{i_1}, \pi^{t'+1}(x_{i_1}) = y_{i_2}, \ldots, \pi^{t'+1}(x_{i_r}) = y_{s+1}$.

We may suppose without loss of generality that the resampling of the bad event first swaps $x_{i_1}, \ldots, x_{i_r}$ in that order. Let π' denote the result of these swaps; there may be additional swaps to other elements of the permutation, but we must have $\pi^{t+1}(x_{i_l}) = \pi'(x_{i_l})$ for $l = 1, \ldots, r$.

In this case, we see that evidently x_{i_1} swapped with x_{i_2}, then x_{i_2} swapped with x_{i_3}, and so on, until eventually x_{i_r} was swapped with $x'' = (\pi^{t'})^{-1} y_{s+1}$. At this point, we have $\pi'(x'') = y_{i_1}$. At the end of this process, we must have $\pi^{t'+1}(x_0) = y_{i_1}$. We claim that we must have $x'' = x_{i_0}$. For, if x'' is subsequently swapped, then we would have $\pi^{t'+1}(x) = y_{i_1}$ where x is one of the source nodes in the current bad-event; but x_0 is at the *top* of the W-configuration and is not a source node.

This implies that we must have $(\pi^{t'})^{-1} y_s = x'' = x_0$; that is, that $\pi^{t'}(x_0) = y_s$. This in turn implies that $\pi^t(x_0) = y_{s+1}$. For, by Proposition 3.1, otherwise we would have encountered a bad-event involving $(x_0, *)$ or $(*, y_{s+1})$; in either case, we would have a predecessor node $(x_0, *)$ or $(*, y_{s+1})$ in G_t, which contradicts that we have a W-configuration.

Proposition 4.2 can be viewed equally as a definition:

DEFINITION 4.4. (ACTIVE CONDITIONS) *We refer to the conditions implied by Proposition 4.2 as the* active conditions *of the graph G_t; we can view them as a set of pairs $Active(G) = \{(x'_1, y'_1), \ldots, (x'_s, y'_s)\}$ such that each (x'_i, y'_i) are the endpoints of a W-configuration of G.*

We also define A^t_k to be the cardinality of $Active(G_t)$, that is, the number of active conditions of permutation π_k at time t. (The subscript k may be omit-

ted in context, as usual.)

When we remove source nodes $(x_1, y_1), \ldots, (x_r, y_r)$ from G_t, the new active conditions of G_{t+1} are related to $(x_1, y_1), \ldots, (x_r, y_r)$ in a particular way.

LEMMA 4.1. *Suppose G is a future-subgraph with source nodes $v_1 \approx (x_1, y_1), \ldots, v_r \approx (x_r, y_r)$. Let $H = G - v_1 - \cdots - v_r$ denote the graph obtained from G by removing these source nodes. Let $Z = \{(x_1, y_1), \ldots, (x_r, y_r)\}$. Then there is a partition of the set Z into two groups $Z = Z_0 \sqcup Z_1$ with the following properties:*

1. *There is an* injective *function $f : Z_1 \to Active(H)$, with the property that $(x, y) \sim f((x, y))$ for all $(x, y) \in Z_1$*
2. *$|Active(H)| = |Active(G)| - |Z_0|$*

Intuitively, we are saying that every node (x, y) we are removing is either explicitly constrained in an "independent way" by some new condition in the graph H (corresponding to Z_1), or it is almost totally unconstrained (corresponding to Z_0). We will never have the bad situation in which a node (x, y) is constrained, but in some implicit way depending on the previous swaps.

Expository remark: *We have recommended bearing in mind the special case when each bad-event consists of a single element. In this case, we would have $r = 1$; and the stated theorem would be that either $|Active(H)| = |Active(G)| - 1$; OR we have $|Active(H)| = |Active(G)|$ and $(x_1, y_1) \sim (x_1', y_1') \in Active(H)$.*

Proof. Let H_i denote the graph $G - v_1 - \cdots - v_i$. We will recursively build up the sets Z_0^i, Z_1^i, f^i, where $\{(x_1, y_1), \ldots, (x_i, y_i)\} = Z_0^i \sqcup Z_1^i$, and which satisfy the given conditions up to stage i.

Now, suppose we remove the source node v_i from H_{i-1}. This will destroy the active condition $(x_i, y_i) \in \text{Active}(H_{i-1})$; it may add or subtract other active conditions as well. We will need to update $Z_0^{i-1}, Z_1^{i-1}, f^{i-1}$. One complication is that f^{i-1} may have mapped (x_j, y_j), for $j < i$, to an active condition of H_{i-1} which is destroyed when v_i is removed. In this case, we must redefine $f^i(x_j, y_j)$ to map to a new active condition.

The first effect of removing v_i from H_{i-1} is to remove (x_i, y_i) from the list of active conditions. Note that we cannot have $f^{i-1}(x_j, y_j) = (x_i, y_i)$ for $j < i$, as the $x_i \neq x_j, y_i \neq y_j$. So far the function f^{i-1} remains a valid mapping from Z_1^{i-1}.

There are now a variety of cases depending on the forward-path of v_i in H_{i-1}.

1. This forward path consists of a cycle, or the forward path terminates on both sides in forward-edges. This is the easiest case. Then no more active conditions of H_{i-1} are created or destroyed. We update $Z_1^i = Z_1^{i-1}, f^i = f^{i-1}$, and $Z_0^i = Z_0^{i-1} \cup \{(x_i, y_i)\}$. One active condition is removed, in net, from H_{i-1}; hence $|\text{Active}(H_i)| = |\text{Active}(H_{i-1})| - 1$.

2. This forward path contains a forward edge on one side and a backward edge on the other. For example, suppose the path has the form $(X_1, Y_1), (X_1, Y_2), (X_2, Y_2), \ldots, (X_s, Y_{s+1})$, where the vertices $(X_1, Y_1), \ldots, (X_s, Y_s)$ are at the base, and the node (X_1, Y_1) has out-degree 1, and the node (X_s, Y_{s+1}) has in-degree 1. Suppose that we remove the source node $(x_i, y_i) = (X_j, Y_j)$ for $1 \leq j \leq s$. (See Figure 2.) In this case, we do not destroy any W-configurations, but we create a new W-configuration with endpoints $(X_j, Y_{s+1}) = (x_i, Y_{s+1})$.

   We now update $Z_1^i = Z_1^{i-1} \cup \{(x_i, y_i)\}, Z_0^i = Z_0^{i-1}$. We define $f^i = f^{i-1}$ plus we map (x_i, y_i) to the new active condition (x_i, Y_{s+1}). In net, no active conditions were added or removed, and $|\text{Active}(H_i)| = |\text{Active}(H_{i-1})|$.

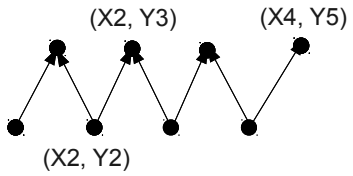

Figure 2: When we remove (X_2, Y_2), we create a new W-configuration with endpoints (X_2, Y_5).

3. This forward path was a W-configuration $(X_0, Y_1), (X_1, Y_1), \ldots, (X_s, Y_s), (X_s, Y_{s+1})$ with $(X_1, Y_1), \ldots, (X_s, Y_s)$ on the base, and we had $(x_i, y_i) = (X_j, Y_j)$. This is the most complicated situation; in this case, we destroy the original W-configuration with endpoints (X_0, Y_{s+1}) but create two new W-configurations with endpoints (X_0, Y_j) and (X_j, Y_{s+1}). We update $Z_0^i = Z_0^{i-1}, Z_1^i = Z_1^{i-1} \cup \{(x_i, y_i)\}$. We will set $f^i = f^{i-1}$, except for a few small changes.

   Now, suppose $f^{i-1}(x_l, y_l) = (X_0, Y_{s+1})$ for some $l < i$; so either $x_l = X_0$ or $y_l = Y_{s+1}$. If it is the former, we set $f^i(x_l, y_l) = (X_0, Y_j), f^i(x_i, y_i) = (X_j, Y_{s+1})$. If it is the latter, we set $f^i(x_l, y_l) = (X_j, Y_{s+1}), f^i(x_i, y_i) =$

(X_0, Y_j).. If $(f^{i-1})^{-1}(X_0, Y_{s+1}) = \emptyset$ then we simply set $f^i(x_i, y_i) = (X_0, Y_j)$.

In any case, f^i is updated appropriately, and in the net no active conditions are added or removed, so we have $|\text{Active}(H_i)| = |\text{Active}(H_{i-1})|$.

5 The probability that the swaps are all successful

In the previous sections, we determined necessary conditions for the permutations π^t, depending on the graphs G_t. In this section, we finish by computing the probability that the swapping subroutine causes the permutations to, in fact, satisfy all such conditions.

Proposition 5.1 states the key randomness condition achieved by the swapping subroutine. The basic intuition behind this is as follows: suppose $\pi : [n] \to [n]$ is a fixed permutation with $\pi(x) = y$, and we call $\pi' = \text{Swap}(\pi; x)$. Then $\pi'(x)$ has a uniform distribution over $[n]$. Similarly, $\pi'^{-1}(y)$ has a uniform distribution over $[n]$. However, the joint distribution is *not* uniform — there is essentially only one degree of freedom for the two values.

PROPOSITION 5.1. *Suppose n, r, s, q are non-negative integers obeying the following constraints:*

1. $0 \le r \le q \le n$
2. $0 \le s \le \min(q, r)$
3. $q + (r - s) \le n$

Let π be a fixed permutation of $[n]$, and let $\pi' = Swap(\pi; x_1, \ldots, x_r)$. Let $(x'_1, y'_1), \ldots, (x'_q, y'_q)$ be a given list with the following properties:

(4) All x' are distinct; all y' are distinct

(5) For $i = 1, \ldots, s$ we have $(x_i, \pi(x_i)) \sim (x'_i, y'_i)$

Then the probability that π' satisfies all the constraints (x', y') is at most

$$P(\pi'(x'_1) = y'_1 \wedge \cdots \wedge \pi'(x'_q) = y'_q) \le \frac{(n-r)!(n-q)!}{n!(n-q-r+s)!}$$

Expository remark: *Consider the special case when each bad-event contains a single element. In that case, we have $r = 1$. There are two possibilities for s; either $s = 0$ in which case this probability on the right is $1 - q/n$ (i.e. the probability that $\pi'(x_1) \neq y'_1, \ldots, y'_q$); or $s = 1$ in which case this probability is $1/n$ (i.e. the probability that $\pi'(x_1) = y'_1$).*

Proof. We can prove this by induction on s, r; this requires a lengthy but elementary argument, which we omit here for reasons of space.

We apply Proposition 5.1 to upper-bound the probability that the Swapping Algorithm successfully swaps when it encounters a bad event.

PROPOSITION 5.2. *Suppose we encounter a bad-event B at time t containing elements (k, x_1, y_1), $\ldots$, (k, x_r, y_r) from permutation k (and perhaps other elements from other permutations). Then the probability that π_k^{t+1} satisfies all the active conditions of its future-subgraph, conditional on all past events and all other swappings at time t, is at most*

$$P(\pi_k^{t+1} \text{ satisfies } Active(G_{k,t+1})) \le P_k(B) \frac{(n_k - A_k^{t+1})!}{(n_k - A_k^t)!}.$$

Recall that we have defined A_k^t to be the number of active conditions in the future-subgraph corresponding to permutation π_k at time t, and we have defined $P_k(B) = \frac{(n_k - r)!}{n_k!}$.

Expository remark: *Consider the special case when each bad-event consists of a single element. In this case, we would have $P_k(B) = 1/n$. The stated theorem is now: either $A^{t+1} = A^t$, in which case the probability that π satisfies its swapping condition is $1/n$; or $A^{t+1} = A^t - 1$; in which case the probability that π satisfies its swapping condition is $1 - A^{t+1}/n$.*

Proof. Let H denote the future-subgraph $G_{k,t+1}$ after removing the source nodes corresponding to $(x_1, y_1), \ldots, (x_r, y_r)$. Using the notation of Lemma 4.1, let $s = |Z_1|$ and let $\text{Active}(H) = \{(x'_1, y'_1), \ldots, (x'_q, y'_q)\}$. where $q = A_k^{t+1}$.

For each $(x, y) \in Z_1$, we have $y = \pi^t(x)$, and there is an injective function $f : Z_1 \to \text{Active(H)}$ and $(x, y) \sim f((x, y))$. We can assume without loss of generality $Z_1 = \{(x_1, y_1), \ldots, (x_s, y_s)\}$ and $f(x_i, y_i) = (x'_i, y'_i)$. In order to satisfy the active conditions on $G_{k,t+1}$, the swapping must cause $\pi^{t+1}(x'_i) = y'_i$ for $i = 1, \ldots, q$.

By Lemma 4.1, we have $n_k \ge A_k^t = A_k^{t+1} + (r - s)$, so all the conditions of Proposition 5.1 are satisfied. Thus this probability is at most $\frac{(n_k - r)!}{n_k!} \times \frac{(n_k - q)!}{(n_k - q - r + s)!} = \frac{(n_k - r)!(n_k - A_k^{t+1})!}{n_k!(n_k - A_k^t)!}$.

We have finally all the pieces necessary to prove Lemma 3.1.

LEMMA 5.1. *Let τ be a witness tree, with nodes labeled $B_1, \ldots, B_s$. For any event E, the probability that τ was produced as the witness tree corresponding to event E, is at most*

$$P(\hat{\tau} = \tau) \le P(B_1) \cdots P(B_s)$$

Proof. Suppose the witness tree τ contains T nodes. Consider a subtree $\tau' \subseteq \tau$, with $T' \le T$ nodes, obtained

by successively removing leaf nodes. We will prove the following by induction on T': The probability, starting at any state of the Swapping Algorithm and for any seed-event E, that the subsequent swaps would produce the subtree τ', is at most

$$P(\hat{\tau}_{\geq T-T'} = \tau') \leq \prod_{B \in \tau'} P(B) \times \prod_{k=1}^{N} \frac{n_k!}{(n_k - |\text{Active}(\text{Proj}_k(\tau'))|)!}.$$

The base case $T' = 0$ is vacuously true. To show the induction step, note that in order for τ' to be produced, some $B \in \tau'$ must at some point be resampled. Now integrate over the first such resampled B, and applying Proposition 5.2 in combination with the induction hypothesis.

We now consider the necessary conditions to produce the *entire* witness tree τ, and not just fragments of it. First, the *original permutations* π_k^0 must satisfy the active conditions of the respective witness dags $\text{Proj}_k(\tau)$. For each permutation k, this occurs with probability $\frac{(n_k - A_k^0)!}{n_k!}$. Next, the subsequent sampling must be compatible with τ; applying the previous result with $T' = T$ (i.e. $\tau' = \tau$) yields that this has probability $\prod_{B \in \tau} P(B) \times \prod_{k=1}^{N} \frac{n_k!}{(n_k - A_k^0)!}$. In total we have

$$P(\hat{\tau} = \tau) \leq \prod_k \frac{(n_k - A_k^0)!}{n_k!} \times \prod_{B \in \tau} P(B) \times \prod_{k=1}^{N} \frac{n_k!}{(n_k - A_k^0)!} = \prod_{B \in \tau} P(B).$$

We note one counter-intuitive aspect to this proof. The natural way of proving this lemma would be to identify, for each bad-event $B \in \tau$, some necessary event occurring with probability at most $P(B)$. This is the general strategy in Moser-Tardos [31] and related constructive LLL variants such as [21]. This is *not* the proof we employ here; there is an additional factor of $(n_k - A_k^0)!/n!$ which is present for the original permutation and is gradually "discharged" as active conditions disappear from the future-subgraphs.

6 The constructive LLL for permutations

Now that we have proved the key lemma regarding witness trees, the remainder of the analysis is essentially the same as for the Moser-Tardos algorithm [31]. Using arguments and proofs from [31] with our key lemma, we can now easily show our key theorem:

THEOREM 6.1. *Suppose there is some assignment of weights $\mu : \mathcal{B} \to [0, \infty)$ which satisfies, for every $B \in \mathcal{B}$ the condition*

$$\mu(B) \geq P(B) \prod_{B' \sim B} (1 + \mu(B'))$$

Then the Swapping Algorithm terminates with probability one. The expected number of iterations in which we resample B is at most $\mu(B)$.

In the "symmetric" case, this gives us the well-known LLL criterion:

COROLLARY 6.1. *Suppose each bad-event $B \in \mathcal{B}$ has probability at most p, and is dependent with at most d bad-events. Then if $ep(d+1) \leq 1$, the Swapping Algorithm terminates with probability one; the expected number of resamplings of each bad-event is $O(1)$.*

Some extensions of the LLL, such as the observation of Pegden regarding independent sets in the dependency graph [34], or the partial-resampling of [21], follow almost immediately here. There are a few extensions which require slightly more discussion, which is done next in Sections 6.1 and 6.2.

6.1 Lopsidependence As in [31], it is possible to slightly restrict the notion of dependence. Two bad-events which share the same valuation of a variable are not forced to be dependent. We can re-define the relation $\sim$ on bad-events as follows: for $B, B' \in \mathcal{B}$, we have $B \sim B'$ iff

1. $B = B'$, *or*
2. there is some $(k, x, y) \in B, (k, x', y') \in B'$ with either $x = x', y \neq y'$ or $x \neq x', y = y'$.

In particular, bad-events which share the same triple (k, x, y), are *not* caused to be dependent.

Proving that the Swapping Algorithm still works in this setting requires only a slight change in our definition of $\text{Proj}_k(\tau)$. Now, the tree τ may have multiple copies of any given triple (k, x, y) on a single level. When this occurs, we create the corresponding nodes $v \approx (x, y) \in \text{Proj}_k(\tau)$; edges are added between such nodes in an arbitrary (but consistent) way. The remainder of the proof remains as before.

6.2 LLL for injective functions The analysis of [27] considers a slightly more general setting for the LLL, in which we select random *injections* $f_k : [m_k] \to [n_k]$, where $m_k \leq n_k$. In fact, our Swapping Algorithm can be extended to this case. We simply define a permutation π_k on $[n_k]$, where the entries $\pi_k(m_k +$

$1), \ldots, \pi_k(n_k)$ are "dummies" which do not participate in any bad-events. The LLL criterion for the extended permutation π_k is exactly the same as the corresponding LLL criterion for the injection f_k. Because all of the dummy entries have the same behavior, it is not necessary to keep track of the dummy entries exactly; they are needed only for the analysis.

7 A parallel version of the Swapping Algorithm

The Moser-Tardos resampling algorithm for the ordinary LLL can be transformed into an RNC algorithm by allowing a slight slack in the LLL's sufficient condition [31]. The basic idea is that in every round, we select a *maximal independent set* of bad-events to resample. Using the known distributed/parallel algorithms for MIS, this can be done in RNC; the number of resampling rounds is then shown to be logarithmic whp ("with high probability"), in [31].

In this section, we will describe a parallel algorithm for the Swapping Algorithm, which runs along the same lines. However, everything is more complicated than in the case of the ordinary LLL. In the Moser-Tardos algorithm, events which are not connected to each other cannot affect each other in any way. For the permutation LLL, such events can interfere with each other, but do so rarely. Consider the following example. Suppose that at some point we have two active bad-events, "$\pi_k(1) = 1$" and "$\pi_k(2) = 2$" respectively, and so we decide to resample them simultaneously (since they are not connected to each other, and hence constitute an independent set). When we are resampling the bad-event $\pi_k(1) = 1$, When we are resampling the bad-event $\pi_k(1) = 1$, we may swap 1 with 2; in this case, we are automatically fixing the second bad-event as well. The sequential algorithm, in this case, would only swap a single element. The parallel algorithm should likewise *not* perform a second swap for the second bad-event, or else it would be over-sampling. Avoiding this type of conflict is quite tricky.

Let $n = n_1 + \cdots + n_K$; since the output of the algorithm will be the contents of the permutations $\pi_1, \ldots, \pi_k$, this algorithm should be measured in terms of n. Hence we must show that this algorithm runs in $\log^{O(1)} n$ time. We will make the following assumptions in this section. First, we assume that $|\mathcal{B}|$, the total number of potential bad-events, is polynomial in n. This assumption can be relaxed if we have the proper kind of "separation oracle" for $\mathcal{B}$. Next, we assume that every element $B \in \mathcal{B}$ has size $|B| \leq M = \log^{O(1)} n$. This holds in all cases that we are aware of.

We describe the following Parallel Swapping Algorithm:

1. In parallel, generate the permutations $\pi_1, \ldots, \pi_N$ uniformly at random.
2. We now go through a series of *rounds* while there is some true bad-event. In round i ($i = 1, 2, \ldots,$) do the following:
    3. Let $\mathcal{V}_{i,1} \subseteq \mathcal{B}$ denote the set of bad-events which are currently true at the beginning of round i. We will attempt to fix the bad-events in $\mathcal{V}_{i,1}$ through a series of *sub-rounds*. This may introduce new bad-events, but we will not fix any newly created bad-events until round $i + 1$.

    We repeat the following for $j = 1, 2, \ldots$ as long as $\mathcal{V}_{i,j} \neq \emptyset$:

        4. Let $I_{i,j}$ be a maximal independent set (MIS) of bad-events in $\mathcal{V}_{i,j}$.
        5. For each true bad-event $B \in I_{i,j}$, choose the swaps corresponding to B. Namely, if we have some bad-event B involving triples $(k_1, x_1, y_1), \ldots, (k_r, x_r, y_r)$, then we select each $z_l \in [n_{k_l}]$, which is the element to be swapped with $\pi_{k_l}(x_l)$ according to procedure Swap. *Do not perform the indicated swaps at this time though!* We refer to $(k_1, x_1), \ldots, (k_r, x_r)$ as the swap-sources of B and the $(k_1, z_1), \ldots, (k_r, z_r)$ as the swap-mates of B.
        6. Select a random ordering $\rho_{i,j}$ of the elements of $I_{i,j}$. Consider the graph $G_{i,j}$ whose vertices correspond to elements of $I_{i,j}$: add an edge connecting B with B' if $\rho_{i,j}(B) < \rho_{i,j}(B')$ *and* one of the swap-mates of B is a swap-source of B'. Generate $I'_{i,j} \subseteq I_{i,j}$ as the *lexicographically-first MIS* (LFMIS) of the resulting graph $G_{i,j}$, with respect to the vertex-ordering $\rho_{i,j}$.
        7. For each permutation π_k, enumerate all the transpositions $(x\ z)$ corresponding to elements of $I'_{i,j}$, arranged in order of $\rho_{i,j}$. Say these transpositions are, in order $(x_1, z_1), \ldots (x_l, z_l)$, where $l \leq n$. Compute, in parallel for all π_k, the composition $\pi'_k = \pi_k(x_l\ z_l) \ldots (x_1\ z_1)$.
        8. Update $\mathcal{V}_{i,j+1}$ from $\mathcal{V}_{i,j}$ by removing all elements which are either no longer true for the current permutation, *or* are connected via $\sim$ to some element of $I'_{i,j}$.

This algorithm can be viewed as simulating the sequential Swapping Algorithm, with a particular choice of selecting which bad-event to resample. Because

our analysis of the sequential Swapping Algorithm was already based on an *arbitrary* choice of which bad-event to resample, we immediately have the proposition:

PROPOSITION 7.1. *Consider the execution log of resampled bad events, ordered by sub-rounds i, j and by the orders $\rho_{i,j}$ within a sub-round. Let $\hat{\tau}$ be the corresponding witness tree. Then*

$$P(\hat{\tau} = \tau) \leq \prod_{B \in \tau} P(B)$$

Proof. As one examines the bad-events in this order, then the choice of swaps for a given true bad-event B depends only on the "past" (events with a smaller value of i, j or ρ.) Hence we can construct a witness tree, ordered by i, j, ρ instead of time, and apply similar probabilistic arguments to the sequential algorithm. It is absolutely critical in this argument that are able to order the bad-events in this way, such that any probabilities of performing a certain swapping for a given bad event depends only on the "past". If in step (6), we had selected $I'_{i,j}$ to be an arbitrary MIS of $I_{i,j}$, then we could no longer guarantee this, and the dependencies would be much harder to control.

We now must analyze the steps of this algorithm and show they can be implemented in parallel polylogarithmic time. We must also show that the total number of sub-rounds is itself polylogarithmic.

Most of the steps of this algorithm can be implemented using standard parallel algorithms. For example, step (1) can be performed simply by having each element of $[n_k]$ choose a random real and then executing a parallel sort. The independent set $I_{i,j}$ can be found in time in polylogarithmic time using [5, 28].

The difficult step to parallelize is in selecting the LFMIS $I'_{i,j}$. In general, the problem of finding the LFMIS is P-complete [11], hence we do not expect a generic parallel algorithm for this. However, what saves us it that the ordering $\rho_{i,j}$ and the graph $G_{i,j}$ are constructed in a highly random fashion.

This allows us to use the following greedy algorithm to construct $I'_{i,j}$, the LFMIS of $G_{i,j}$:

1. Let H_1 be the directed graph obtained by orienting all edges of $G_{i,j}$ in the direction of $\rho_{i,j}$. Repeat the following for $s = 1, 2, \ldots,$:
2. If $H_s = \emptyset$ terminate.
3. Find all source nodes of H_s. Add these to $I'_{i,j}$.
4. Construct H'_{s+1} by removing all source nodes and all successors of source nodes from H'_s.

The output of this algorithm is the LFMIS $I'_{i,j}$. Each step can be implemented in parallel time $O(1)$. The number of iterations of this algorithm is the length of the longest directed path in $G'_{i,j}$. So it suffices it show that, whp, all directed paths in $G'_{i,j}$ have length at most polylogarithmic in n.

PROPOSITION 7.2. *Let $I \subseteq \mathcal{B}$ be an an arbitrary independent set of true bad-events, and suppose all elements of $\mathcal{B}$ have size $\leq M$.*

Generate ρ as a random ordering of I and for each element of I generate the associated swap-mates. Form the directed graph on G in which there is an edge from B to B' iff $\rho(B) < \rho(B')$ and one of the swap-mates of B is a swap-source of B'.

Then whp, every directed path in G has length $O(M + \log n)$.

Proof. If $|I| \leq 5(M + \log n)$ this is clear, so we suppose $|I| > 5(M + \log n)$ in this proof. One of the main ideas below is to show that for the *typical* $B_1, \ldots, B_l \in I$, where $l = 5(M + \log n)$, the probability that $B_1, \ldots, B_l$ form a directed path is small.

Suppose we select $B_1, \ldots, B_l \in I$ uniformly at random without replacement. Let us analyze how these could form a directed path in G.

First, it must be the case that $\rho(B_1) < \rho(B_2) < \cdots < \rho(B_l)$. This occurs with probability $1/l!$.

Next, it must be that the swap-mates of B_s overlap the swap-sources of B_{s+1}, for $s = 1, \ldots, l-1$. Now, B_s has $O(M)$ swap-mates; each such swap-mate can overlap with at most one element of I, since I is an independent set. Conditional on having chosen $B_1, \ldots, B_s$, there are a remaining $|I| - s$ choices for B_{s+1}. This gives that the probability of having B_s with an edge to B_{s+1}, conditional on the previous events, is at most $\frac{M}{|I|-s}$. (The fact that swap-mates are chosen randomly does not give too much of an advantage here.)

Putting this all together, the total probability that there is a directed path on $B_1, \ldots, B_l$ is

$$P(\text{directed path } B_1, \ldots, B_l) \leq \frac{M^{l-1}(|I| - l)!}{(|I| - 1)! l!}$$

Since the above was for a random $B_1, \ldots, B_l$, the probability that there is *some* such path (of length l) is at most

$$\begin{aligned} P(\text{some directed path}) &\leq \frac{|I|!}{(|I| - l)!} \times \frac{M^{l-1}(|I| - l)!}{(|I| - 1)! l!} \\ &= |I| \times \frac{M^{l-1}}{l!} \\ &\leq n \times \frac{M^{l-1}}{(l/e)^l} \end{aligned}$$

$$\leq n^{-\Omega(1)},$$

since $l = 5(M + \log n)$.

So far, we have shown that each sub-round of the Parallel Swapping Algorithm can be executed in parallel time $\log^{O(1)} n$. Next, we show that whp that number of sub-rounds corresponding to any round is bounded by $\log^{O(1)} n$.

PROPOSITION 7.3. *Suppose $|\mathcal{B}| = n^{O(1)}$ and all elements $B \in \mathcal{B}$ have size $|B| \leq M$. Then whp, we have $\mathcal{V}_{i,j} = \emptyset$ for some $j = O(M \log n)$.*

Proof. We will first show the following: Let $B \in I$, where I is an arbitrary independent set of $\mathcal{B}$. Then with probability at least $1 - \frac{1-e^{-M}}{M}$ we have $B \in I'$ as well, where I' is the LFMIS associated with I.

Suppose B involves r_k elements from permutations $k = 1, \ldots, N$. As I is an independent set, each element of $[n_k]$ is involved in at most one element of I. Suppose that the other elements of I which intersect permutation k involve respectively $r'_{k,1}, \ldots, r'_{k,s}$ elements from that permutation. Note that $r_k + (r'_{k,1} + \cdots + r'_{k,s}) \leq n_k$.

Then B will be put into I' if none of the swap-mates of those other elements, which have a smaller value of ρ, overlap with B in that permutation. We say that in this case B is *undominated in permutation k*.

We will analyze the ordering ρ using the standard trick, in which each element $B \in I$ chooses a rank $W(B) \sim \text{Uniform}[0, 1]$, independently and identically. The ordering ρ is then formed by sorting in increasing ordering of W. In this way, we are able to avoid the dependencies induced by the rankings. For the moment, let us suppose that the rank $W(B)$ is *fixed* at some real value w.

Now consider a permutation k, and consider some bad-event B' which contains $r'_{k,l}$ elements from that permutation k. The probability that B is undominated by B' through permutation k is

$$\begin{aligned} &P(B \text{ undominated by } B' \mid W(B) = w) \\ &\quad = (1 - w) + w\frac{(n_k - r_k)!(n_k - r'_{k,l})!}{n_k!(n_k - r_k - r'_{k,l})!} \\ &\quad \geq 1 - \frac{wr'_{k,l}r_k}{n - r'_{k,l} + 1} \end{aligned}$$

Here, the factor of w reflects the fact that, conditional on a fixed value for $W(B) = w$, the element B' has $\rho(B') < \rho(B)$ with probability w.

All such events are independent (conditional on $W(B) = w$); this implies that the total probability that B is undominated in permutation k is at least

$$\begin{aligned} &P(B \text{ undominated in permutation } k \mid W(B) = w) \\ &\quad \geq \prod_{l=1}^{s}\Big(1 - \frac{wr'_{k,l}r_k}{n - r'_{k,l} + 1}\Big) \end{aligned}$$

The quantity $1 - \frac{wr'_{k,l}r_k}{n-r'_{k,l}+1}$ is log-concave in $r'_{k,l}$; hence this product is minimized when there are $s = n_k - r_k$ other elements with $r'_{k,1}, \ldots, r'_{k,s} = 1$. This yields

$$\begin{aligned} &P(B \text{ undominated in permutation } k \mid W(B) = w) \\ &\quad \geq \Big(1 - \frac{wr_k}{n_k}\Big)^{n_k - r_k} \\ &\quad \geq e^{-r_k w} \end{aligned}$$

Take the product over all permutations k, bearing in mind that $\sum_k r_k = |B| \leq M$. This gives us that the total probability that B is undominated, conditional on $W(B) = w$, is at least e^{-Mw}. Now integrate over $W(B)$; this gives us

$$\begin{aligned} P(B \text{ undominated in permutation } k) &\geq \int_{w=0}^{1} e^{-Mw} dw \\ &= \frac{1 - e^{-M}}{M} \end{aligned}$$

Now, using this fact, we show that $\mathcal{V}_{i,j}$ is decreasing quickly in size. For, suppose $B \in \mathcal{V}_{i,j}$. So $B \sim B'$ for some $B' \in I_{i,j}$, as $I_{i,j}$ is a maximal independent set (possibly $B = B'$). We will remove B from $\mathcal{V}_{i,j+1}$ if $B' \in I'_{i,j}$, which occurs with probability at least $1 - \frac{1-e^{-M}}{M}$. As B was an arbitrary element of $\mathcal{V}_{i,j}$, this shows that $\mathbf{E}\big[|\mathcal{V}_{i,j+1}| \mid \mathcal{V}_{i,j}\big] \leq (1 - \frac{1-e^{-M}}{M})|\mathcal{V}_{i,j}|$.

For $j = \Omega(M \log n)$, this implies that

$$\mathbf{E}\big[|\mathcal{V}_{i,j}|\big] \leq (1 - \frac{1 - e^{-M}}{M})^{\Omega(M \log n)}|\mathcal{V}_{i,1}| \leq n^{-\Omega(1)}$$

This in turn implies that $\mathcal{V}_{i,j} = \emptyset$ with high probability, for $j = \Omega(M \log n)$.

To finish the proof, we must show that the number of rounds is itself bounded whp. For this we have the following proposition:

PROPOSITION 7.4. *Let B be any resampling performed at the ith round of the Parallel Swapping Algorithm (that is, $B \in I'_{i,j}$ for some integer $j > 0$.) Then the witness tree corresponding to the resampling of B has height exactly i.*

Proof. First, note that if we have $B \sim B'$ in the execution log, where B occurs earlier in time, and the

witness tree corresponding to B has height i, then the witness tree corresponding to B' must have height $i+1$. So it will suffice to show that if $B \in I'_{i,j}$, then we must have $B \sim B'$ for some $B' \in I'_{i-1,j'}$.

At the beginning of round i, it must be the case that π^i makes the bad-event B true. By Proposition 3.1, either the bad-event B was already true at the beginning of round $i-1$, or some bad-event $B' \sim B$ was swapped at round $i-1$. If it is the latter, we are done.

So suppose B was true at the beginning of round $i-1$. So B was an element of $\mathcal{V}_{i-1,1}$. In order for B to have been removed from $\mathcal{V}_{i-1}$, then either we had $B \sim B' \in I'_{i-1,j'}$, in which case we are also done, or after some sub-round j' the event B was no longer true. But again by Proposition 3.1, in order for B to become true again at the beginning of round i, there must have been some bad-event $B' \sim B$ encountered later in round $i-1$.

This gives us the key bound on the running time of the Parallel Swapping Algorithm. We give only a sketch of the proof, since the argument is identical to that of [31].

THEOREM 7.1. *Suppose $|\mathcal{B}| = n^{O(1)}$ and that for all $B \in \mathcal{B}'$ we have $|B| \leq \log^{O(1)} n$.*

Suppose also that $\epsilon > 0$ and that there is some assignment of weights $\mu : \mathcal{B} \to [0, \infty)$ which satisfies, for every $B \in \mathcal{B}$, the condition

$$\mu(B) \geq (1+\epsilon)P(B) \prod_{B' \sim B} (1 + \mu(B'))$$

Then, whp, the Parallel Swapping Algorithm terminates after $\frac{\log^{O(1)} n}{\epsilon}$ rounds. The total time for these rounds is $\frac{\log^{O(1)} n}{\epsilon}$.

Proof. This proof is essentially identical to the corresponding result in [31].

8 Algorithmic Applications

The LLL for permutations plays a role in diverse combinatorial constructions. Using our algorithm, nearly all of these constructions become algorithmic. We examine a few selected applications now.

8.1 Latin transversals Suppose we have an $n \times n$ matrix A. The entries of this matrix come from a set C which are referred to as *colors*. A *Latin transversal* of this matrix is a permutation $\pi \in S_n$, such that no color appears twice among the entries $A(i, \pi(i))$; that is, there are no $i \neq j$ with $A(i, \pi(i)) = A(i', \pi(i'))$. A typical question in this area is the following: suppose each color c appears at most Δ times in the matrix. How large can Δ be so as to guarantee the existence of a Latin transversal?

In [16], a proof using the probabilistic form of the Lovász Local Lemma for permutations was given, showing that $\Delta \leq n/(4e)$ suffices. This was the first application of the LLL to permutations. This bound was subsequently improved by [9] to the criterion $\Delta \leq (27/256)n$; this uses a variant of the probabilistic Local Lemma which is essentially equivalent to Pegden's variant on the constructive Local Lemma. Using our algorithmic LLL, we can almost immediately transform the existential proof of [9] into a constructive algorithm. To our knowledge, this is the first polynomial-time algorithm for constructing such a transversal.

THEOREM 8.1. *Suppose $\Delta \leq (27/256)n$. Then there is a Latin transversal of the matrix. Furthermore, the Swapping Algorithm selects such a transversal in polynomial time.*

Proof. For any quadruples i, j, i', j' with $A(i,j) = A(i',j')$, we have a bad-event $(i,j),(i',j')$. Such an event has probability $\frac{1}{n(n-1)}$. We give weight $\mu(B) = \alpha$ to every bad event B, where α is a scalar to be determined.

This bad-event can have up to four types of neighbors (i_1, j_1, i'_1, j'_1), which overlap on one of the four coordinates i, j, i', j'; as discussed in [9], all the neighbors of any type are themselves neighbors in the dependency graph. Since these are all the same, we will analyze just the first type of neighbor, one which shares the same value of i, that is $i_1 = i$. We now may choose any value for j_1 (n choices). At this point, the color $A(i_1, j_1)$ is determined, so there are $\Delta - 1$ remaining choices for i'_1, j'_1.

By Lemma 3.1 and Pegden's criterion [34], a sufficient condition for the convergence of the Swapping Algorithm is that

$$\alpha \geq \frac{1}{n(n-1)}(1 + n(\Delta - 1)\alpha)^4$$

Routine algebra shows that this has a positive real root α when $\Delta \leq (27/256)n$.

In [39], Szabó considered a generalization of this question: suppose that we seek a transversal, such that no color appears more than s times. When $s = 1$, this is asking for a Latin transversal. Szabó gave similar criteria "$\Delta \leq \gamma_s n$" for s a small constant. Such bounds can be easily obtained constructively using the permutation LLL as well.

By combining the permutation LLL with the partial resampling approach of [21], we can provide asymptotically optimal bounds for large s:

THEOREM 8.2. *Suppose $\Delta \leq (s - c\sqrt{s})n$, where c is a sufficiently large constant. Then there is a transversal of the matrix, in which each color appears no more than s times. This transversal can be constructed in polynomial time.*

Proof. For each set of s appearances of any color, we have a bad event. We use the partial resampling framework, to associate the fractional hitting set which assigns weight $\binom{s}{r}^{-1}$ to any r appearances of a color, where $r = \lceil \sqrt{s} \rceil$.

We first compute the probability of selecting a given r-set X. From the fractional hitting set, this has probability $\binom{s}{r}^{-1}$. In addition, the probability of selecting the indicated cells is $\frac{(n-r)!}{n!}$. So we have $p \leq \binom{s}{r}^{-1} \frac{(n-r)!}{n!}$.

Next, we compute the dependency of the set X. First, we may select another X' which overlaps with X in a row or column; the number of such sets is $2rn\binom{\Delta}{r-1}$. Next, we may select any other r-set with the same color as X (this is the dependency due to $\bowtie$ in the partial resampling framework; see [21] for more details). The number of such sets is $\binom{\Delta}{r}$.

So the LLL criterion is satisfied if

$$e \times \binom{s}{r}^{-1} \frac{(n-r)!}{n!} \times \left(2rn\binom{\Delta}{r-1} + \binom{\Delta}{r}\right) \leq 1$$

Simple calculus now shows that this can be satisfied when $\Delta \leq (s - O(\sqrt{s}))n$. Also, it is easy to detect a true bad-event and resample it in polynomial time, so this gives a polynomial-time algorithm.

Our result depends on the Swapping Algorithm in a fundamental way — it does not follow from Theorem 1.1 (which would roughly require $\Delta \leq (s/e)n$). Hence, prior to this paper, we would not have been able to even show the existence of such transversals; here we provide an efficient algorithm as well. To see that our bound is asymptotically optimal, consider a matrix in which the first $s+1$ rows all contain a given color, a total multiplicity of $\Delta = (s+1)n$. Then the transversal must contain that color at least $s+1$ times.

8.2 Strong chromatic number of graphs Suppose we have a graph G, with a given partition of the vertices into k blocks each of size b, i.e., $V = V_1 \sqcup \cdots \sqcup V_k$. We would like to b-color the vertices, such that every block has exactly b colors, and such that no edge has both endpoints with the same color (i.e., it is a proper vertex-coloring). This is referred to as a *strong coloring* of the graph. If this is possible for *any* such partition of the vertices into blocks of size b, then we say that the graph G has strong chromatic number b.

A series of papers [3, 8, 18, 22] have provided bounds on the strong chromatic number of graphs, typically in terms of their maximum degree Δ. In [23], it is shown that when $b \geq (11/4)\Delta + \Omega(1)$, such a coloring exists; this is the best bound currently known. Furthermore, the constant $11/4$ cannot be improved to any number strictly less than 2. The methods used in most of these papers are highly non-constructive, and do not provide algorithms for generating such colorings.

In this section, we examine two routes to constructing strong colorings. The first proof, based on [1], builds up the coloring vertex-by-vertex, using the ordinary LLL. The second proof uses the permutation LLL to build the strong coloring directly. The latter appears to be the first RNC algorithm with a reasonable bound on b.

We first develop a related concept to the strong coloring known as an *independent transversal*. In an independent transversal, we choose a single vertex from each block, so that the selected vertices form an independent set of the graph.

PROPOSITION 8.1. *Suppose $b \geq 4\Delta$. Then G has an independent transversal, which can be found in expected time $O(n\Delta)$.*

Furthermore, let $v \in G$ be any fixed vertex. Then G has an independent transversal which includes v, which can be found in expected time $O(n\Delta^2)$.

Proof. Use the ordinary LLL to select a single vertex uniformly from each block. See [9], [21] for more details. This shows that, under the condition $b \geq 4\Delta$, an independent transversal exists and is found in expected time $O(n\Delta)$.

To find an independent transversal including v, we imagine assigning a weight 1 to vertex v and weight zero to all other vertices. As described in [21], the expected weight of the independent transversal returned by the Moser-Tardos algorithm, is at least $\Omega(w(V)/\Delta)$, where $w(V)$ is the total weight of all vertices. This implies that that vertex v is selected with probability $\Omega(1/\Delta)$. Hence, after running the Moser-Tardos algorithm for $O(\Delta)$ separate independent executions, one finds an independent transversal including v.

Using this as a building block, we can form a strong coloring by gradually adding colors:

THEOREM 8.3. *Suppose $b \geq 5\Delta$. Then G has a strong coloring, which can be found in expected time $O(n^2\Delta^2)$.*

Proof. (This proof is almost identical to the proof of Theorem 5.3 of [1]). We maintain a *partial coloring* of the graph G, in which some vertices are colored with $\{1, \ldots, b\}$ and some vertices are uncolored. Initially all

vertices are uncolored. We require that in a block, no vertices have the same color, and no adjacent vertices have the same color. (This only applies to vertices which are actually colored).

Now, suppose some color is partially missing from the strong coloring; say without loss of generality there is a vertex w missing color 1. In each block $i = 1, \ldots, k$, we will select some vertex v_i to have color 1. If the block does not have such a vertex already, we will simply assign v_i to have color 1. If the block i *already* had some vertex u_i with color 1, we will swap the colors of v_i and u_i (if v_i was previously uncolored, then u_i will become uncolored).

We need to ensure three things. First, the vertices $v_1, \ldots, v_k$ must form an independent transversal of G. Second, if we select vertex v_i and swap its color with u_i, this cannot cause u_i to have any conflicts with its neighbors. Third, we insist of selecting w itself for the independent traversal.

A vertex u_i will have conflicts with its neighbors if v_i currently has the same color as one of the neighbors of u_i. In each block, there are at least $b-\Delta$ possible choices of v_i that avoid that; we must select an independent transversal among these vertices, which also includes the designated vertex w. By Proposition 8.1, this can be done in time $O(n^2\Delta^2)$ as long as $b \geq 4\Delta$.

Whenever we select the independent transversal $v_1, \ldots, v_k$, the total number of colored vertices increases by at least one: for, the vertex w becomes colored while it was not initially, and in every other block the number of colored vertices does not decrease. So, after n iterations, the entire graph has a strong coloring; the total time is $O(n^2\Delta^2)$.

The algorithm based on the ordinary LLL is slow and is inherently sequential. Using the permutation LLL, one can obtain a more direct and faster construction; however, the hypothesis of the theorem will need to be slightly stronger.

THEOREM 8.4. *Suppose we have a given graph G of maximum degree Δ, whose vertices are partitioned into blocks of size b. Then if $b \geq \frac{256}{27}\Delta$, it is possible to strongly color graph G in expected time $O(n\Delta)$. If $b \geq (\frac{256}{27} + \epsilon)\Delta$ for some constant $\epsilon > 0$, there is an RNC algorithm to construct such a strong coloring.*

Proof. We will use the permutation LLL. For each block, we assume the vertices and colors are identified with the set $[b]$. Then any proper coloring of a block corresponds to a permutation of S_b. When we discuss the color of a vertex v, we refer to $\pi_k(v)$ where k is the block containing vertex v.

For each edge $f = \langle u, v\rangle \in G$ and any color $c \in [1, \ldots b]$, we have a bad-event that both u and v have color c. (Note that we cannot specify simply that u and v have the *same color*; because we have restricted ourselves to *atomic* bad-events, we must list every possible color c with a separate bad event.)

Each bad-event has probability $1/b^2$. We give weight $\mu(B) = \alpha$ to every bad event, where α is a scalar to be determined.

Now, each such event (u, v, c) is dependent with four other types of bad-events:

1. An event u, v', c' where v' is connected to vertex u;
2. An event u', v, c' where u' is connected to vertex v;
3. An event u', v', c where u' is in the block of u and v' is connected to u';
4. An event u', v', c where v' is in the block of v and u' is connected to v'

There are $b\Delta$ neighbors of each type. For any of these four types, all the neighbors are themselves connected to each other. Hence an *independent* set of neighbors of the bad-event (u, v, c) can contain one or zero of each of the four types of bad-events.

Using Lemma 3.1 and Pegden's criterion [34], a sufficient condition for the convergence of the Swapping Algorithm is that

$$\alpha \geq (1/b^2) \cdot (1 + b\Delta\alpha)^4$$

When $b \geq \frac{256}{27}\Delta$, this has a real positive root α^* (which is a complicated algebraic expression). Furthermore, in this case the expected number of swaps of each permutation is $\leq b^2\Delta\alpha^* \leq \frac{256}{81}\Delta$. So the Swapping Algorithm terminates in expected time $O(n\Delta)$. A similar argument applies to the parallel Swapping Algorithm.

8.3 Acyclic edge-coloring We now consider a classical coloring problem that is notable because there is no obvious permutation in its description; however, our framework still proves useful.

Given a graph G, the *acyclic edge-coloring problem* is to produce a proper edge-coloring with a "small" number C of colors, with the additional property that there are no even-length cycles in G in which the edges receive exactly two colors. Introduced by Grünbaum in 1973 [19], this problem has received a good deal of attention [4, 7, 17, 30, 32, 33]. Letting Δ denote the maximum degree of G, recall that $\Delta \leq C \leq \Delta + 1$ if we only wanted a proper edge-coloring; thus, much work on the problem has studied C as a function of Δ. The best bound currently known, due to [17], is that $C = 4\Delta-4$ is admissible; this bound is also constructive. The algorithm of [17] is based on choosing colors for the edges in a *sequential* fashion, to systematically avoid

all repeated edge-colors and 4-cycles. Cycles of length 6 or more cannot be avoided in this way; for these, a resampling-based approach similar to [31] is employed. This algorithm seems difficult to parallelize because the possible colors of an edge are dependent on its neighbors.

Older algorithms for this problem are based on the ordinary (asymmetric) LLL; each edge chooses a color independently, and conflicts are resampled. The most recent paper to use this analysis is [33]; this shows that the LLL and the algorithm of [31] produce a coloring when $C \geq 9.62\Delta$. These algorithms are easy to parallelize using [31].

We apply the permutation LLL to show that $C \geq 8.42\Delta + O(1)$ suffices. This almost directly leads to an RNC algorithm that achieves the fewest colors of any known RNC algorithm; again, the notable feature is that there is no obvious "permutation" in the problem description:

THEOREM 8.5. *Suppose Δ is sufficiently large and $C \geq 8.42\Delta$. Then the graph G has an acyclic edge-coloring with C colors. Furthermore, if C, Δ are constants satisfying this condition, then there is an RNC algorithm to find it.*

Proof. We assume that Δ is sufficiently large through the course of this proof, and include an ϵ slack in our bounds on C. This simplifies many of the bounds and arguments.

The construction has two parts. First, one assigns each edge to randomly "belong" to one of its two endpoints. This can be done using the ordinary LLL. It is not hard to see that we can assign edges to vertices such that each vertex owns at most $\Delta' \leq (1+\epsilon)\Delta/2$ edges.

After this is done, each vertex chooses a random injection from the Δ' edges it owns to the set of c colors. The idea here is that we systematically prevent two edges incident to a vertex v from sharing a color, *if* they are both owned by the vertex v. It is possible that one or both of these edges are owned by the other vertices; in this case, we must resample those edges. These types of conflicts cannot be systematically avoided.

We two types of bad-events. In the first type, two edges incident on the same vertex share a color c_1. In the second type, there is a cycle of length $2l \geq 4$, in which the colors alternate between c_1 and c_2. For a bad-event of the first type we define $\mu(B) = \alpha$ and for any bad-event B of the second type which depends on $2l$ edges, we define $\mu(B) = \alpha^l$, where α is a constant to be determined.

Now, given a fixed edge $f \in G$ and fixed color c, one can verify that the sum of $\mu(B)$, over all bad-events B which contain $\pi(f) = c$, is at most

$$t \leq 3\Delta'\alpha + (C-1)\sum_{l=2}^{\infty} \alpha^l (2\Delta')^{2l-2}$$
$$= 3\Delta'\alpha + \frac{4\alpha^2(C-1)(\Delta')^2}{1-4\alpha(\Delta')^2}$$

The presence of the constant 3 is notable here; this is because it is impossible for two edges owned by the same vertex to have the same color. Using the ordinary LLL, this constant would instead be 4.

Given a bad-event B of length $2l$, we use Pegden's criterion to count the sum of $\mu(B')$ over the neighborhoods of B. B contains $2l$ edges, each with a specified color. For each combination of edge and color, there may be a single child with the same edge (but possibly a different color), and there may be a single child using the same color for the originating vertex (but assigned to a different edge). Hence, for such an event B, we have

$$\sum_{\substack{\text{independent sets } X \\ \text{of neighbors of } B}} \prod_{B' \in X} \mu(X) \leq ((1+Ct)(1+\Delta' t))^{2l}$$

This gives us the following LLL criterion: for any bad-event B of length $2l$, we must have

$$P(B) \leq \frac{\alpha^l}{((1+Ct)(1+\Delta' t))^{2l}}$$

The probability of B is at most $(C-1)^{-2l}$; so this is satisfied for all B if

$$1 \leq \frac{(C-1)^2\alpha}{((1+Ct)(1+\Delta' t))^2}$$

Routine algebraic manipulations show that this has a real positive solution $\alpha > 0$ when $C \geq 16.8296\Delta'$. For Δ sufficiently large, this is satisfied when $C \geq 8.42\Delta$.

This does not immediately give us an algorithm to construct such a coloring, because the number of potential bad-events is not polynomially bounded in size. However, as discussed in [20], one can truncate the set of cycles under consideration to length $O(\log n)$. With high probability, longer cycles will have no bad-events, even if we do not explicitly check them.

Now, the short cycles satisfy the conditions of Theorem 7.1, so the Parallel Swapping Algorithm finds an acyclic edge-coloring with high probability in $\log^{O(1)} n$ rounds.

9 Conclusion

The algorithm we have developed in this paper appears to be a special case of a more general paradigm. Namely,

suppose we want to assign variables $X_1, \ldots, X_n$ to satisfy some constraint satisfaction problem. These constraints have two types: there are "local constraints," which are isolated, sporadic constraints that depend on only a few variables at a time; there are also "global constraints," which are large-scale and highly structured. A random approach works well to satisfy the local constraints, but totally fails for the global constraints. In this case, a compromise is needed: we choose the variables $X_1, \ldots, X_n$ in a structured way so that the global constraints are automatically satisfied. Now there may still be a few isolated local constraints left unsatisfied, and so we need to perform some small modifications to fix these.

While this paper has examined global constraints of the form $X_i \neq X_j$ for certain values of i, j, the local lemma has been applied to a variety of other probability distributions. The works [26, 29] survey a range of probabilistic settings in which the lopsided LLL may be applied, which includes hypergraph matchings, set partitions, and spanning trees. Another example comes from [14], which applies an LLL variant to the space of Hamiltonian cycles.

This type of algorithm requires a "local correction" step that satisfies two properties. First, it introduces enough randomness so that local constraints may be satisfied probabilistically; second, it only makes a small change to the variables, so that constraints which were already satisfied are not too disrupted. Note that these properties are in conflict: we could simply resample all of the affected variables, which introduces the maximal amount of randomness as well as disruption.

The algorithm of [17] for acyclic edge-colorings is especially notable in this regard, because it uses this paradigm of algorithms in a setting *not directly covered by any form of the LLL.*

We may ask the question: what types of global structures admit this form of resampling algorithm? What are the precise criteria on the randomness and locality of the resampling step?

Acknowledgment. We thank the referees for their detailed and helpful comments.

References

[1] Aharoni, R., Berger, E., Ziv, R.: Independent systems of representatives in weighted graphs. Combinatorica 27.3, pp. 253-267 (2007).

[2] Alon, N.: Probabilistic proofs of existence of rare events. Springer Lecture Notes in Mathematics No. 1376 (J. Lindenstrauss and V. D. Milman, Eds.), Springer-Verlag, pp. 186-201 (1988).

[3] Alon, N.: The strong chromatic number of a graph. Random Structures and Algorithms 3-1, pp. 1-7 (1992).

[4] Alon, N., McDiarmid, C., Reed, B.: Acyclic coloring of graphs. Random Structures & Algorithms, 2(3):277–288 (2007).

[5] Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. J. Algorithms 7(4): 567-583 (1986).

[6] Alon, N., Spencer, J., Tetali, P.: Covering with Latin transversals. Discrete Applied Math. 57, pp. 1-10 (1995).

[7] Alon, N., Sudakov, B., Zaks, A.: Acyclic edge colorings of graphs. Journal of Graph Theory, 37(3):157–167 (2001).

[8] Axenovich, M., Martin, R.: On the strong chromatic number of graphs. SIAM J. Discrete Math 20-3, pp. 741-747 (2006).

[9] Bissacot, R., Fernandez, R., Procacci, A., Scoppola, B.: An improvement of the Lovász Local Lemma via cluster expansion. Combinatorics, Probability and Computing 20-5, pp. 709-719 (2011).

[10] Bottcher, J., Kohayakawa, Y., Procacci, A.: Properly coloured copies and rainbow copies of large graphs with small maximum degree. Random Structures and Algorithms 40-4, pp. 425-436 (2012).

[11] Cook, S.: A Taxonomy of problems with fast parallel algorithms. Information and Control 64, pp. 2-22 (1985).

[12] Dénes, J., Keedwell, A. D.: Latin squares and their applications. Akadémiai Kiadó, Budapest & English Universities Press (1974).

[13] Hahn, G., Thomassen, C.: Path and cycle sub-Ramsey numbers and an edge-colouring conjecture. Discrete Mathematics 62-1, pp. 29-33 (1986)

[14] Dudek, A., Frieze, A., Rucinski, A.: Rainbow Hamilton cycles in uniform hypergraphs. The Electronic Journal of Combinatorics 19-1: P46 (2012)

[15] Erdős, P., Hickerson, D. R., Norton, D. A., Stein, S. K.: Has every Latin square of order n a partial Latin transversal of size n-1? Amer. Math. Monthly 95, pp. 428430 (1988).

[16] Erdős, P., Spencer, J.: Lopsided Lovász Local Lemma and Latin transversals. Discrete Applied Math 30, pp. 151-154 (1990).

[17] Esperet, L., Parreau, A.: Acyclic edge-coloring using entropy compression. European Journal of Combinatorics 34-6, pp. 1019-1027 (2013).

[18] Fellows, M.: Transversals of vertex partitions in graphs. SIAM J. Discrete Math 3-2, pp. 206-215 (1990).

[19] Grünbaum, B.: Acyclic colorings of planar graphs. Israel Journal of Mathematics, 14:390–408 (1973).

[20] Haeupler, B., Saha, B., Srinivasan, A.: New constructive aspects of the Lovász Local Lemma. Journal of the ACM 58 (2011).

[21] Harris, D., Srinivasan, A.: The Moser-Tardos framework with partial resampling. Proc. IEEE Symp. Foundations of Computer Science, 2013.

[22] Haxell, P.: On the strong chromatic number. Combi-

natorics, Probability, and Computing 13-6, pp. 857-865 (2004).
[23] Haxell, P.: An improved bound for the strong chromatic number. Journal of Graph Theory 58-2, pp. 148-158 (2008).
[24] Keevash, P., Ku, C.: A random construction for permutation codes and the covering radius. Designs, Codes and Cryptography 41-1, pp. 79-86 (2006).
[25] Kolipaka, K., Szegedy, M.: Moser and Tardos meet Lovász. Proc. ACM Symposium on Theory of Computing, pp. 235-244 (2011).
[26] Lu, L., Mohr, A., Szekely, L.: Quest for negative dependency graphs. Recent Advances in Harmonic Analysis and Applications pp. 243-258 (2013).
[27] Lu, L., Szekely, L.: Using Lovász Local Lemma in the space of random injections. The Electronic Journal of Combinatorics 13-R63 (2007).
[28] Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15(4):1036-1053 (1986).
[29] Mohr, A.: Applications of the Lopsided Lovász Local Lemma regarding hypergraphs. PhD Thesis, University of South Carolina (2013).
[30] Molloy, M., Reed, B.: Further algorithmic aspects of the Local Lemma. Proc. ACM Symposium on Theory of Computing, pp. 524-529 (1998).
[31] Moser, R., Tardos, G.: A constructive proof of the general Lovász Local Lemma. Journal of the ACM 57-2, pp. 11:1-11:15 (2010).
[32] Muthu, R., Narayanan, N., Subramanian, C. R.: Improved bounds on acyclic edge colouring. Discrete Mathematics, 307(23):3063–3069 (2007).
[33] Ndreca, S., Procacci, A., Scoppola, B.: Improved bounds on coloring of graphs. European Journal of Combinatorics 33-4, pp. 592-609 (2012).
[34] Pegden, W.: An extension of the Moser-Tardos algorithmic Local Lemma. Arxiv 1102.2583 (2011).
[35] Scott, A., Sokal, A.: The repulsive lattice gas, the independent-set polynomial, and the Lovász Local Lemma. J. Stat. Phys. 118, No. 5-6, pp. 11511261 (2005).
[36] Shearer, J. B.: On a problem of Spencer. Combinatorica 5, 241-245 (1985).
[37] Shor, P. W.: A lower bound for the length of a partial transversal in a Latin square. J. Combin. Theory Ser. A. 33, pp. 1-8 (1982).
[38] Stein, S. K.: Transversals of Latin squares and their generalizations. Pacific J. Math. 59, pp. 567-575 (1975).
[39] Szabó, S.: Transversals of rectangular arrays. Acta Math. Univ. Comenianae, Vol. 37, pp. 279-284 (2008).