On the constructive Lovász Local Lemma

Tan Likai, Wang Zhijian, Ryan Chew

April 2020

1 Introduction

Often in probabilistic analysis, we want to show existence of (and possibly construct) an "ideal" object. However, we have a set of "bad" events in our probability space, that we want to avoid. To this end, the union bound and independence conditions are probability tools that are often used to bound the probability of failure.

However, independence is an extremely strong condition and is in general not true in most cases. On the other hand, the union bound is often too weak to produce meaningful results.

The Lovász Local Lemma (LLL) is a probabilistic tool which guarantees an existence of such an ideal object when the dependence between events is **limited**.

Theorem 1.1 (Symmetric Lovász Local Lemma)**.** *Let $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ be a set of events in a probability space Ω. Suppose each event is dependent on at most d other events, and occurs with probability at most p. If $ep(d+1) \leq 1$, then* $\Pr\left[\overline{A_1} \wedge \overline{A_2} \wedge ... \wedge \overline{A_n}\right] > 0$.

When the events are not symmetrically bounded, there is an alternate version of the Lovász Local Lemma:

Theorem 1.2 (Asymmetric Lovász Local Lemma)**.** *Let $\mathcal{A} = \{A_1, A_2, ..., A_n\}$ be a set of events in a probability space Ω. For each event $A \in \mathcal{A}$, let $\Gamma(A)$ be the set of events that are not mutually independent with A. If we can assign reals $x(A) \in (0,1)$ for each event A such that for all $A \in \mathcal{A}$,*

$$\Pr[A] < x(A) \prod_{B \in \Gamma(A)} (1 - x(B)),$$

then $\Pr\left[\overline{A_1} \wedge \overline{A_2} \wedge ... \wedge \overline{A_n}\right] > 0$.

We will not prove the non-constructive versions here as that is not the focus of our report. It is however of note that the original proof by Lovász and Erdős is non-constructive. Therefore, even though the existence of the "ideal" object is guaranteed, there was no efficient method to find it. Due to its application in the probabilistic method, there was a significant incentive to devise an algorithm that could find this object efficiently.

2 Algorithmic Lovász Local Lemma

In [1], Beck demonstrated an algorithmic version of the LLL. However, this formulation had a weaker degree bound of $|\Lambda(A)| < 2^{\frac{n}{48}}$, compared to the nonconstructive lemma's tight bound of $|\Lambda(A)| < \frac{2^n}{e}$. Later, a result by Moser and Tardos in [2], which was built upon Moser's previous work a year ago, tightened this bound further to match the original bound.

In Moser's constructive version of the LLL, we add the constraint that we work over a finite set of mutually independent random variables $\mathcal{P}$ in our sample space Ω. Then, each bad event A_i is determined by some subset of these variables $\text{vbl}(A_i) \subseteq \mathcal{P}$. Without loss of generality, we take the unique minimal such subset. Given an evaluation (assignment of values to variables) of $\mathcal{P}$, if $A \in \mathcal{A}$ occurs, we say that this evaluation causes A to be violated. Note that two events $A, A' \in \mathcal{A}$ are independent if and only if $\text{vbl}(A) \cap \text{vbl}(A') = \emptyset$.

Algorithm 1 Moser's algorithm

```
function SOLVE($\mathcal{P}, \mathcal{A}$)
    for $P \in \mathcal{P}$ do
        $v_P \leftarrow$ a random evaluation of $P$
    end for
    while $\exists A_i \in \mathcal{A}$ violated do
        for $P \in \text{vbl}(A_i)$ do                              ▷ Resample $P$
            $v_P \leftarrow$ a new random evaluation of $P$
        end for
    end while
    return $(v_P)_{P \in \mathcal{P}}$
end function
```

Moser's algorithm is a randomized resampling algorithm that constructs an assignment to $\mathcal{P}$, such that no A_i is violated.

Theorem 2.1 (Algorithmic Lovász Local Lemma)**.** *Let $\mathcal{P}$ be a finite set of mutually independent random variables in a probability space Ω. Let $\mathcal{A}$ be a finite set of events determined by these variables. If there exists an assignment of reals to events $x : \mathcal{A} \to \mathbb{R}$, such that*

$$\forall A \in \mathcal{A} : P(A) \leq x(A) \prod_{B \in \Gamma(A)} (1 - x(B))$$

then there exists an assignment of values to the variables $\mathcal{P}$ not violating any event $A \in \mathcal{A}$. Furthermore, the algorithm resamples an event $A \in \mathcal{A}$ at most an expected $\frac{x(A)}{1-x(A)}$ times, before finding such an assignment. Thus, the total number of sampling steps is at most $\sum_{A \in \mathcal{A}} \frac{x(A)}{1-x(A)}$ in expectation.

2.1 Execution Logging

Moser's proof of runtime hinges on the idea of recording the execution of the algorithm. It logs the resampled bad event $C(t) \in \mathcal{A}$ for each time $t \geq 1$. This is equivalent to a (possibly partial) function $C : \mathbb{N} \to \mathcal{A}$.

Before defining the log, it is first necessary to define a few terms. For an event $C \in \mathcal{A}$, its *neighborhood* $\Gamma(C)$ is defined to be the set of events that are not independent with C. Its *inclusive neighborhood* $\Gamma^+(C)$ is the union $\Gamma(C) \cup C$.

This execution log is used to build a **witness tree** for each resampling step, which contains the entire history of events which led to this event being resampled at this time.

A witness tree is a finite rooted tree, where each vertex corresponds to a resampling step. We label every vertex v with the resampled event $[v] \in \mathcal{A}$ corresponding to that step. Now, we enforce that two adjacent vertices in the witness tree, are labelled by events adjacent in the dependency graph. This captures the idea that a preceding resampling step could have produced an evaluation, which caused the current event to be violated. A proper witness tree is a witness tree, such that each vertex has children with distinct event labels.

Now, given an execution log C, we build an witness tree $\tau_C(t)$ for each step. This tree represents the resampling steps and corresponding bad events, which are (possibly indirectly) dependent with the current resampled bad event.

We start with the root node, and label it with the current event $C(t)$. Then, iteratively from $t' = t-1, t-2, \ldots, 2, 1$, we examine the t'-th resampling. Given the resampled event $C(t')$, if it is in the current set of bad events in the tree, or adjacent to them in the dependency graph, we add it to our tree. Specifically, given all vertices $v \in V(\tau_C(t))$ that satisfy $C(t') \in \Gamma^+([v])$, we pick the one with greatest depth from the root (breaking ties arbitrarily). Then, we add t' as its child (labelled with $C(t')$).

The first main claim of Moser's papers (Lemma 2.1(i) of [2] and Lemma 2.2 of his original paper) states that any intermediate witness tree generated at some resampling step will be a proper witness tree. Also, it is easy to see that the same witness tree will never appear twice in the log, since by considering the number of times the event at the root vertex has been resampled, all witness trees in C with a common root are distinct.

Moser's approach was to bound the expected number of resamplings by bounding the number of distinct witness trees, which was in turn bounded by the probabilities that each witness tree would appear in the log.

We have the following lemma (Lemma 2.1(ii) of [2]):

Lemma 2.1 (A probability bound on witness trees)**.** *For a proper witness tree τ, the probability of the witness tree appearing in C is at most $\prod_{v\in V(\tau)} \Pr[[v]]$.*

Proof. We give a slightly different proof from Moser and Tardos, but the main idea is the same. The idea behind the proof is to replay the algorithm using a stream of random samples of variables in $\mathcal{P}$, known as the *assignment table.*

For each resampling done by the algorithm, the variables in the violated event are resampled according to the assignment table. For τ to appear in C, for each vertex $v \in V(\tau)$, the state of $\mathrm{vbt}([v])$ must have been such that the bad event $[v]$ is violated. We compute the probability that this occurs.

Consider the vertices in τ in decreasing order of depth. If $d(u) < d(v)$, then either u must occur after v in the algorithm, or $[u]$ and $[v]$ are independent. If $d(u) = d(v)$, $[u]$ and $[v]$ are independent, otherwise the one resampled earlier would have been assigned as a child of the other.

For each $v \in V(\tau)$, consider the *pre-state*, i.e. the random assignment of variables in $\mathrm{vbl}([v])$ immediately prior to the resampling for v. To appear in C, These variables are obtained from the assignment table. The following two claims show some properties of the pre-states.

Claim 1. *The position of the variables in the assignment table for each pre-state is fixed for a given witness tree τ.*

Proof. For a vertex $v \in V(\tau)$ of some depth k and some $P \in \mathrm{vbl}([\mathrm{v}])$, consider the number of times N where P was resampled by vertices of depth greater than k. No other resamples of P can occur before v, since no vertex of the same depth depends on P, no other prior events will resample P (otherwise they will be in the witness tree with depth greater than k), and any vertex that has lesser depth and resamples P must occur after P by definition of witness tree. Therefore the pre-state contains exactly the $(N + 1)$-th sample of P. Hence the position of the pre-state of v is fixed. □

Claim 2. *The pre-states of different vertices are independent of each other, i.e. they share no common variable in the assignment table.*

Proof. Prove by contradiction. Suppose that the pre-states of u and v contain the same sample for some variable P. From the previous claim, the pre-state of v contains the $(N + 1)$-th sample for P where N is the number of resamples from vertices of greater depth. If $d(u) < d(v)$, let N' be the number of resamples from vertices of depth between $d(v)$ and $d(u) + 1$. Since v resamples P, $N' \geq 1$. Then the pre-state of u contains the $(N' + N + 1)$-th sample of P, a different sample from the pre-state of v. By symmetry, $d(u) > d(v)$ is not possible as well. So $d(u) = d(v)$. This is a contradiction since $[u]$ and $[v]$ are dependent but events of vertices at the same layer are independent. □

Within the assignment table, consider the variables contained in the pre-states of the vertices in $V(\tau)$. For a vertex $v \in V(\tau)$, the probability that the pre-state violates $[v]$ is $\Pr[[v]]$. Note that τ appears in C only if all pre-states violate their corresponding events. Since the pre-states are mutually independent, the probability of τ appearing in C is at most $\prod_{v\in V(\tau)} \Pr[[v]]$. This finishes up the proof for Lemma 2.1. □

Now, we want to count the number of times a bad event A was resampled, N_A. For each resampling, it corresponds to a distinct proper witness tree, with the root vertex labelled A. Hence, it is equivalent to a sum of indicator variables over all proper witness trees with root A. We define a helper function

$x'(A)$:

$$\begin{aligned}
x' &: A \to \mathbb{R} \\
x'(A) &= x(A) \prod_{B \in \Gamma(A)} (1 - x(B)) \\
\mathbb{E}[N_A] &= \sum_{\tau \in \mathcal{T}_A} \mathbb{E}[I(\tau \text{ appears in log } C)] = \sum_{\tau \in \mathcal{T}_A} P(\tau \text{ appears in log } C) \\
&\leq \sum_{\tau \in \mathcal{T}_A} \prod_{v \in V(\tau)} \Pr[[v]] \qquad \text{(By Lemma 2.1)} \\
&\leq \sum_{\tau \in \mathcal{T}_A} \prod_{v \in V(\tau)} x'([v]) \qquad \text{(By assumption in Theorem 2.1)}
\end{aligned}$$

We want to relate each $\prod_{v \in V(\tau)} \Pr[[v]]$ term to $x(A)$. Moser constructs a multitype Galton-Watson branching process, for generating any proper witness tree with root A. Initially, we start with a root vertex labelled A. Then, in every round, we consider only newly added vertices from the previous round. For every new vertex labelled $[v]$ in the current tree, we consider its possible child labels $B \in \Gamma^+([v])$. We add a child vertex with label B with probability $x(B)$, and skip it with probability $1 - x(B)$. All these events are independent.

Moser then proves the following lemma (Lemma 3.1 in [2]).

Lemma 2.2. *Let τ be a fixed proper witness tree with its root vertex labelled A. The probability p_τ that the Galton-Watson process described above yields exactly the tree t is*

$$p_\tau = \frac{1 - x(A)}{x(A)} \prod_{v \in V(\tau)} x'([v])$$

$$\begin{aligned}
\mathbb{E}[N_A] &\leq \sum_{\tau \in \mathcal{T}_A} \prod_{v \in V(\tau)} x'([v]) \\
&= \sum_{\tau \in \mathcal{T}_A} \frac{x(A)}{1 - x(A)} p_\tau = \frac{x(A)}{1 - x(A)} \sum_{\tau \in \mathcal{T}_A} p_\tau \\
&\leq \frac{x(A)}{1 - x(A)}
\end{aligned}$$

as the summation of p_τ corresponding to disjoint events, is at most 1. This proves the upper bound on the expected number of resamples of event A in Moser's algorithm.

3 Weakening the Conditions

The previous proof by Moser put bounds on the number of resampling steps needed, given the bound on $P(A)$ parameterized by $x(A)$. For example, for the problem of k-SAT, each non-contradictory clause has at least 2^{-k} probability of not being satisfied by a uniform random assignment (saturated iff the variables are distinct). Here, we often use the symmetric version of the LLL with dependence degree d. We set $x(A) = \frac{1}{d+1}$, and produce a satisfying assignment of variables. Rearranging the inequality,

$$d \leq \frac{2^k}{e} - 1$$

We will now investigate the behaviour of the number of steps, as the dependence degree d goes past that threshold. We assume a satisfying assignment exists, despite not being guaranteed by the LLL. We can slightly relax the constraint on $P(A)$ by adding a $(1+\epsilon)$ factor.

$$\begin{aligned}
P(A) &\le (1+\epsilon)x'(A) \le \frac{1+\epsilon}{e(d+1)} \\
P(\tau \text{ appears in } \log C) &\le \prod_{v\in V(\tau)} \Pr[[v]] \qquad \text{(By Lemma 2.1)} \\
&\le \prod_{v\in V(\tau)} (1+\epsilon)x'([v]) \qquad \text{(By weaker assumption)} \\
&= (1+\epsilon)^{|V(\tau)|} \prod_{v\in V(\tau)} x'([v]) \\
&= (1+\epsilon)^{|V(\tau)|} \frac{x(A)}{1-x(A)} p_\tau \\
\mathbb{E}[N_A] &\le \frac{x(A)}{1-x(A)}\, \mathbb{E}_{\tau\sim GW}[(1+\epsilon)^{|V(\tau)|}] \\
&= \frac{x(A)}{1-x(A)} G_{|V(\tau)|}(1+\epsilon)
\end{aligned}$$

where $G(s)$ is the PGF for $|V(\tau)|$ in Moser's Galton-Watson distribution.

If we assume the dependency graph of the k-SAT instance is d-regular, we can model (upper bound) the child node distribution as $\text{Bin}(d+1,q)$, with a PGF of $G_{\text{Bin}}(s) = (1-p+sp)^{d+1}$, and q is the maximum of all $x(A)$.

We split the generated tree into groups of vertices by generation. Let $Z_n, n \ge 0$ be the number of vertices in generation n.

$$Z_0 = 1$$

Let X_k be the number of children of the k-th vertex in generation n. We assumed all vertices have the same child distribution.

$$\begin{aligned}
Z_{n+1} &= \sum_{k=1}^{Z_n} X_k \\
G_{Z_{n+1}} &= \sum_{z=0}^{\infty} P(Z_{n+1}=z)s^z \\
&= \sum_{z=0}^{\infty}\sum_{y=0}^{\infty} P(Z_{n+1}=z \mid Z_n=y)P(Z_n=y)s^z \\
&= \sum_{y=0}^{\infty} P(Z_n=y)\sum_{z=0}^{\infty} P(Z_{n+1}=z \mid Z_n=y)s^z
\end{aligned}$$

Conditioned on $Z_n=y$, Z_{n+1} is the sum of y independent copies of $\text{Bin}(d+1,q)$. Hence, this is just the product of y copies of $G_{\text{Bin}}(s)$.

$$= \sum_{y=0}^{\infty} P(Z_n=y)G_{\text{Bin}}(s)^y$$

We now have a similar summation to $G_{Z_n}(y)$, but with $G_{\text{Bin}}(s)^y$ instead of s^y.

$$= G_{Z_n}(G_{\text{Bin}}(s))$$

Now we want to compute the expected size of each generation.

$$\begin{aligned}
\mathbb{E}_{X\sim\text{Bin}(d+1,q)}[X] &= G'_{\text{Bin}}(1) = q(d+1) = \mu \\
\mathbb{E}[Z_0] &= 1 \\
\mathbb{E}[Z_{n+1}] &= G'_{Z_{n+1}}(1) \\
&= (\frac{d}{ds} G_{Z_n}(G_{\text{Bin}}(s)))_{s=1} \\
&= G'_{Z_n}(G_{\text{Bin}}(1))G'_{\text{Bin}}(1) \\
&= G'_{Z_n}(1)\mu \\
&= \mu^{n+1}
\end{aligned}$$

The total size of the tree is:

$$\begin{aligned}
T &= Z_0 + Z_1 + Z_2 + \dots \\
\mathbb{E}[T] &= 1 + \mu + \mu^2 + \mu^3 + \dots
\end{aligned}$$

If $\mu = q(d+1) > 1$, then $\mathbb{E}[T]$ diverges. Then, our original expectation $\mathbb{E}[(1+\epsilon)^T]$ diverges as well:

$$\begin{aligned}
\mathbb{E}[(1+\epsilon)^T] &= \mathbb{E}[1 + \epsilon T + O((\epsilon T)^2)] \\
&\geq 1 + \epsilon\,\mathbb{E}[T] \to \infty
\end{aligned}$$

Hence, for the symmetric LLL, the original GW method in Moser's paper fails to give any bounds past the critical threshold $q(d+1) > 1$. This is the case for k-SAT. In actuality, given an execution log, all witness trees consistent with it must be finite. So a potential question to answer would be the actual extent how the bound on the number of execution steps of Moser's algorithm breaks down once the threshold is exceeded.

4 Experiment

The behavior of Moser's algorithm beyond the threshold is investigated by running the algorithm on multiple classes of solvable k-SAT instances. During every run of the algorithm, we record the total number of resampling steps performed by the algorithm.

All source code and test data used in the experiment can be found on `https://github.com/zj-cs2103/cs5330-project`.

4.1 Generation of k-SAT Instances

Two algorithms were employed to generate k-SAT instances for running tests on Moser's algorithm, explained below.

4.1.1 k-SAT Instances with fixed dependence degree

Solvable k-SAT instances with a fixed dependence degree d were generated in MiniSat format. In summary, the desired dependence degree d, the number of literals in each clause k, the number of distinct literals and the number of clauses is supplied to the algorithm as parameters. Then, the algorithm chooses a random clause and a random literal, and add the variable to the clause as long as adding the literal to the clause does not violate the condition on d. Finally, all clauses are padded with extra literals to ensure that there are exactly k literals per clause, and all literals were negated with probability 1/2. The instances were then verified to be solvable using MiniSat.

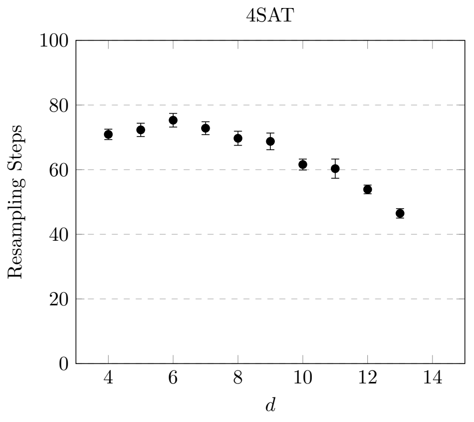
4.1.2 ToughSAT algorithm for hard k-SAT instances

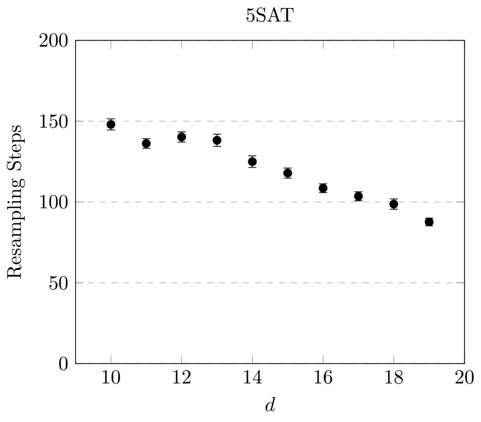
The ToughSAT algorithm (by Yuen and Bebel, `https://toughsat.appspot.com/`) was employed to generate solvable, but more difficult k-SAT instances. In summary, for each clause, a random subset of size k of the available literals are chosen to form the clause. Each literal is then negated with probability 1/2. The ToughSAT algorithm does not take the dependence degree d as a parameter and is hence able to produce more difficult instances. All instances were verified to be solvable using MiniSat.

4.2 Empirical Results

4.2.1 Initial Tests

Moser's algorithm was tested on 100 random 4SAT and 5SAT instances with 100 clauses and 70 literals, as well as 200 clauses and 130 literals respectively. On each instance, Moser's algorithm was run for 100 times, and the average number of iterations was taken. We examine the behaviour of the total number of resampling steps performed by Moser's algorithm on 4SAT and 5SAT instances as the dependence degree d is brought past the threshold.





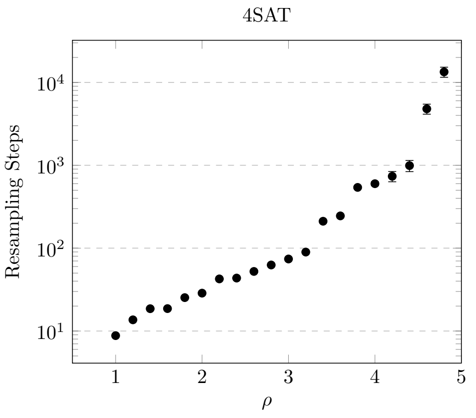
Surprisingly, the empirical results indicate that Moser's algorithm performs well for 4SAT instances where d is three times past the threshold of 4, and for 5SAT instances where d is two times past the threshold of 10. We also observe that there is a slight decrease in the number of resampling steps with an increase in d. This may be due to the generation of the k-SAT clauses: by keeping the number of variables and clauses fixed while increasing the dependence degree, there might be a decrease in the difficulty of the k-SAT instances.

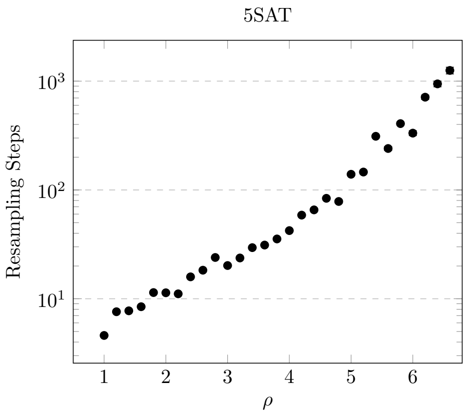
With the observations above, one question is to what extent Moser's algorithm will continue to perform well if we continue to increase the dependence degree.

4.2.2 Tests for k-SAT Instances with Dense Dependency Graphs

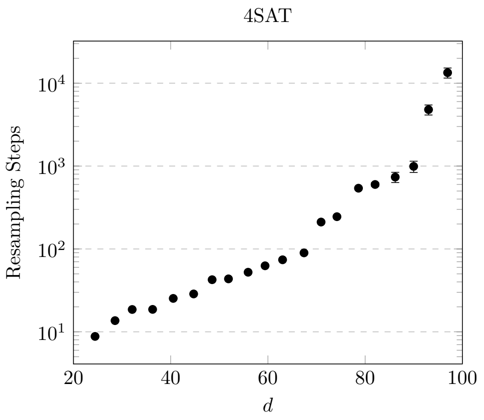
We observe that d is a measure of how dense the dependency graph of a k-SAT instance is. High values of d lead to relatively dense dependency graphs, and low values of d lead to relatively sparse dependency graphs. Here, we introduce a slightly different measure of how dense a dependency graph is that follows closely with how d measures the density of a dependency graph. We define the density of a dependency graph ρ to be the ratio of the number of clauses to the number of literals. For a fixed number of literals, a higher values of ρ indicates that a greater number of clauses are required to 'share' the literals, therefore, it is more likely for clauses to share common literals with each other, and thus implies a relatively higher value of d.

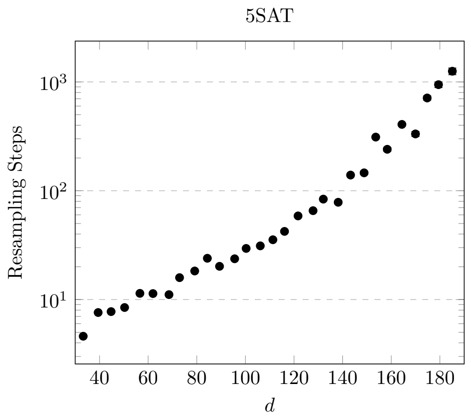
Moser's algorithm was run on 4SAT and 5SAT instances with 100 literals. The number of clauses in each instance was varied, and the number of resampling steps used by Moser's algorithm was once again measured.





We also compute the average value of d for each ρ used in the testing. Below, we graph the number of resampling steps taken by Moser's algorithm against the average value of d for each value of ρ.





We observe that Moser's algorithm shows good performance up to $\rho = 3.2$ for the case of 4SAT, and $\rho = 4.8$ for the case of 5SAT, with the number of resampling steps taken increasing exponentially beyond. This corresponds to an average value of $d = 67$ and $d = 138$ respectively. Compared to the thresholds of 4 and 10 for 4SAT and 5SAT respectively, we see that on Moser's algorithm performs well on random SAT instances way beyond the threshold. One possible reason for this is that Lovász Local Lemma is generally used to prove the existence of bad events, instead of determining the exact performance of the algorithm.

Another observation we make is that though we are experimenting with just the ratio of the number of clauses to the number of literals, Moser's algorithm shows better performance on random instances of 5SAT as compared to 4SAT, as a lower number of resampling steps is used in general for the same value of ρ. This indicates that the value of k in the SAT instances that Moser's algorithm is executed on may play a role in the overall number of resampling steps as well.

5 Conclusion

We have analyzed the constructive Lovász Local Lemma, as well as how the number of resampling steps performed by Moser's algorithm varies as the dependence degree d goes past the threshold given by the Lovász Local Lemma. We have also shown that empirically, Moser's algorithm performs well beyond the threshold given by the Lovász Local Lemma on random k-SAT instances.

References

[1] József Beck. "An algorithmic approach to the Lovász local lemma. I". In: *Random Structures & Algorithms* 2.4 (1991), pp. 343–365. DOI: `10.1002/rsa.3240020402`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.3240020402`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.3240020402`.

[2] Robin A. Moser and Gábor Tardos. “A constructive proof of the general Lovasz Local Lemma”. In: *CoRR* abs/0903.0544 (2009). arXiv: `0903.0544`. URL: `http://arxiv.org/abs/0903.0544`.