Class 12
Algorithmic LLL
Announcements

• HW5 due Friday
• HW6 out now!
• HW7 isn’t due until after fall break! (Friday 12/2)
• No class on Tuesday: Democracy Day!

If you are eligible to vote, then VOTE!
Recap: Algorithmic LLL (for k-SAT)

• Given \( \varphi \):
  • Choose a random assignment \( \sigma \) for each of the variables that appear in \( \varphi \)
  • While there is some clause \( C \) of \( \varphi \) that is not satisfied:
    • Update \( \sigma \) by randomly re-selecting the variables that appear in \( C \).
  • Return \( \sigma \)

• Theorem:
  • Suppose that each clause \( C \) in \( \varphi \) shares variables with at most \( d + 1 = 2^{k-c} \) clauses (including \( C \) itself), for some constant \( c \).
  • Then \( \varphi \) is satisfiable and the algorithm above finds a satisfying assignment quickly.
Algorithmic LLL more generally

• Given $V$ and $\mathcal{A}$:
  • Choose a random assignment $\sigma_v$ for each of the random variables $v \in V$
  • While there is some $A \in \mathcal{A}$ so that $A(\sigma) = 1$:
    • Choose (arbitrarily) an event $A$ with $A(\sigma) = 1$.
    • Update $\sigma$ by re-selecting $\{\sigma_v : v \in \text{Vbl}(A)\}$ randomly.

• Suppose that for all $A \in \mathcal{A}$:
  • $|\Gamma(A)| \leq d + 1$
  • $\Pr[A] \leq \frac{1}{e(d+1)}$

• Then this algorithm will find an assignment to the variables in $V$ so that no event of $\mathcal{A}$ occurs with $O\left(\frac{|\mathcal{A}|}{d+1}\right)$ re-randomizations.

$\mathcal{A}$ is a collection of bad events determined by variables in $V$. Vbl($A$) is the set of variables involved with $A \in \mathcal{A}$
Proof of Algorithmic LLL

- Add some print statements to our algorithm.
- If the algorithm runs for too long, it will be too good of a compression algorithm.
Questions?
Algorithmic LLL, Quiz?
Q1: Applying alg. LLL

• $S_1, S_2, ..., S_M \subset X$ are sets of size $k < |X| = N$
• Each $S_i$ intersects at most 10 other sets $S_j$
• Color points of $X$ red or blue iid with prob $\frac{1}{2}$.
• $A_i$ is the event that $S_i$ is monochromatic.

• $|V| =$
• $d =$
• For what $k$ does alg. LLL apply?
• What is expected number of re-randomizations?
Q2. Changing the proof

• What if we print “trying to fix clause $i_{\ell}$” instead of “trying to fix the $\ell$’th child”?

• Q2.1 How many bits get outputted in print statements?

• Q2.2. What would that prove?
Today: More practice with the Algorithmic LLL

• We saw the proof for k-SAT
• Today you’ll prove it for set coloring!
The problem

• $n$ points, $\{1,2, \ldots, n\}$
• $m$ sets, $S_1, S_2, \ldots, S_m \subseteq \{1, 2, \ldots, n\}$
• Each set has size $k$.
• Each set overlaps with no more than $d$ other sets.
• Goal: color the $n$ points red or blue so that none of the sets is monochromatic.
The problem

- $n$ points, $\{1, 2, \ldots, n\}$
- $m$ sets, $S_1, S_2, \ldots, S_m \subseteq \{1, 2, \ldots, n\}$
- Each set has size $k$.
- Each set overlaps with no more than $d$ other sets.
- Goal: color the $n$ points red or blue so that none of the sets is monochromatic.
Algorithmic LLL gives an algorithm to do this

• While not done:
  • Pick a monochromatic set, $S_i$.
  • Re-color all of the numbers in $S_i$, uniformly at random.

• But we didn’t prove that this works.
  • We only proved it for k-SAT

• Goal of today:
  • Mimic the k-SAT argument to give an algorithm that provably works for no-monochromatic-coloring.
Quick recap of the proof idea for k-SAT

- We wrote the algorithm in a recursive way and added some print statements.
- From the print statements, you could figure out the random bits that went into the algorithm.
- If the algorithm runs for too long (too many re-randomizations), then we can compress the random bits!
- But that’s impossible.
Group work!

• Give a proof!
  • What is the same between the k-SAT proof and this proof?
  • What needs to change?

Outline:
• We wrote the algorithm in a recursive way and added some print statements.
• From the print statements, you could figure out the random bits that went into the algorithm.
• If the algorithm runs for too long (too many re-randomizations), then we can compress the random bits!
• But that’s impossible.
For inspiration, here was the k-SAT algorithm
Your job: adapt to set-coloring!

• **FindSat**$(\varphi = C_1 \land C_2 \land \ldots \land C_m)$:
  • Choose a random assignment $\sigma$ for each of the variables that appear in $\varphi$
  • For each clause $C_i$ in $\varphi$ that is not satisfied:
    • $\sigma \leftarrow \text{Fix}(\varphi, i, \sigma)$
    • Return $\sigma$

• **Fix**$(\varphi, i, \sigma)$:
  • Update $\sigma$ by re-randomizing every variable that appears in the clause $C_i$
  • Let $C_{i_1}, C_{i_2}, \ldots C_{i_{d+1}}$ be the clauses that share variables with $C_i$
  • For $j = 1, \ldots, d + 1$:
    • If $C_{i_j}$ is violated:
      • $\sigma \leftarrow \text{Fix}(\varphi, i_j, \sigma)$
    • Return $\sigma$

All done with this level.

Fixing set $i$!

Trying to fix the $j$'th child

After $T$ re-randomizations, I give up. I've got $\sigma$
What needs to change?

• **FindSat**$(\varphi = C_1 \land C_2 \land \cdots \land C_m)$:
  • Choose a random assignment $\sigma$ for each of the variables that appear in $\varphi$
  • For each clause $C_i$ in $\varphi$ that is not satisfied:
    • $\sigma \leftarrow \text{Fix}(\varphi, i, \sigma)$
    • Return $\sigma$

• **Fix**$(\varphi, i, \sigma)$:
  • Update $\sigma$ by re-randomizing every variable that appears in the clause $C_i$
  • Let $C_{i_1}, C_{i_2}, \ldots C_{i_{d+1}}$ be the clauses that share variables with $C_i$
  • For $j = 1, \ldots, d + 1$:
    • If $C_{i_j}$ is violated:
      • $\sigma \leftarrow \text{Fix}(\varphi, i_j, \sigma)$
    • Return $\sigma$

Fixing set $i$!

Trying to fix the $j$'th child

All done with this level.

After $T$ re-randomizations, I give up. I've got $\sigma$
Our algorithm?

• **FindSat**($S_1, S_2, ..., S_m$):
  • Choose a random **coloring** $\sigma$ for each of **numbers**
  • For each $S_i$ that is monochromatic:
    • $\sigma \leftarrow \text{Fix}(i, \sigma)$
    • Return $\sigma$

• **Fix**(i, $\sigma$):
  • Update $\sigma$ by re-randomizing every **number in** $S_i$
  • Let $S_{i_1}, S_{i_2}, ..., S_{i_{d+1}}$ be the sets that intersect $S_i$
  • For $j = 1, ..., d + 1$:
    • If $S_{i_j}$ is **monochromatic**:
      • $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
    • Return $\sigma$

Fixing set $i$!

Trying to fix the $j$’th child

After $T$ re-randomizations,

I give up. I’ve got $\sigma$
To do the proof

• We need to count the number of random bits that go in in the first $T$ re-randomizations.

• We need to count the number of bits of print statements that come out in the first $T$ re-randomizations.

• We need to argue that we can recover the random bits that go in from the print statements that come out.
To do the proof

• We need to count the number of random bits that go in in the first $T$ re-randomizations.

• We need to count the number of bits of print statements that come out in the first $T$ re-randomizations.

• We need to argue that we can recover the random bits that go in from the print statements that come out.

Whoops! This doesn’t hold!!
Recovering the random bits example

• The print statements allow us to reconstruct the recursion tree.
• Then...

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
Recovering the random bits
example

• The print statements allow us to reconstruct the recursion tree.
• Then…

Trying to fix the $j$'th child

Say we know the coloring AFTER we re-randomized to fix the $j$’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
Recovering the random bits
example

• The print statements allow us to reconstruct the recursion tree.
• Then...

Say we know the coloring AFTER we re-randomized to fix the j’th child.
(We know the final assignment since it was printed out, and we’re working backwards.)
Recovering the random bits example

- The print statements allow us to reconstruct the recursion tree.
- Then...

Since I know the recursion tree, I know that at this point “the j’th child” means $S_4$.

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
Recovering the random bits

example

• The print statements allow us to reconstruct the recursion tree.
• Then...

Since I know the recursion tree, I know that at this point “the j’th child” means $S_4$.

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
Our algorithm

- **FindSat**($S_1, S_2, \ldots, S_m$):
  - Choose a random coloring $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**($i, \sigma$):
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, \ldots S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, \ldots, d + 1$:
    - If $S_{i_j}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
      - Return $\sigma$

All done with this level.

After $T$ re-randomizations, I give up. I’ve got $\sigma$...
...because it was all red! (or blue, as appropriate)

Fixing set $i$!
Recovering the random bits example

- The print statements allow us to reconstruct the recursion tree.
- Then...

Since I know the recursion tree, I know that at this point “the j’th child” means $S_j$.

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
Recovering the random bits
example

• The print statements allow us to reconstruct the recursion tree.
• Then...

Now we know what this assignment was, so we can keep working backwards!

Since I know the recursion tree, I know that at this point “the j’th child” means $S_4$

...because it was all red!

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
To do the proof

- We need to count the number of random bits that go in in the first $T$ re-randomizations.
- We need to count the number of bits of print statements that come out in the first $T$ re-randomizations.
- We need to argue that we can recover the random bits that go in from the print statements that come out.

```
Algorithm
```

```
Random bits  
Formula $\varphi$  
Assignment $\sigma$  
Print statements
```
Our algorithm

- **FindSat**($S_1, S_2, \ldots, S_m$):
  - Choose a random coloring $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**($i, \sigma$):
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, \ldots S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, \ldots, d + 1$:
    - If $S_{i_j}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
    - Return $\sigma$

Random bits in:

- $n + k \cdot T$

Trying to fix the $j$’th child... because it was all red! (or blue, as appropriate)

All done with this level.

Fixing set $i$!

After $T$ re-randomizations, I give up. I’ve got $\sigma$
Our algorithm

- **FindSat**($S_1, S_2, ..., S_m$):
  - Choose a random coloring $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**($i, \sigma$):
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, ..., S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, ..., d + 1$:
    - If $S_{ij}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(ij, \sigma)$
    - Return $\sigma$

Bits out:

$$\leq m \left[ \log(m) + C \right]$$

"Fixing clause $i$"

$$+ T \left[ \log(d+1) + 1 + C \right]$$

"trying to fix $j$th child because it was red"

Also "all done"

$$+ n + C$$

"I give up. $\sigma$"

All done with this level.

Trying to fix the $j$’th child

...because it was all red! (or blue, as appropriate)

After $T$ re-randomizations,

I give up. I’ve got $\sigma$
Win if random bits in $\gg$ bits out
aka, then we'd get a contradiction and conclude that there must be $< T$
re-randomizations.

Random bits in:

$\sigma + k \cdot T$
- original $\sigma$
- $k$ bits per re-randomization

Bits out:

$\leq m[\log(m) + c] + T[\log(d+1) + 1 + c] + n + c$
- "Fixing clause i"
- "trying to fix $j$th child because it was red"
- "I give up. $\delta$."
- call Fix $T$ times
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$ re-randomizations.

Random bits in: $n + k \cdot T$
- $n$: original $\sigma$
- $k \cdot T$: $k$ bits per re-randomization

Bits out:
- $\leq m[\log(m) + C] + T[\log(d+1) + 1 + C] + n + C$
  - “Fixing clause $i$”
  - Trying to fix $j$th child because it was red
  - I give up. $\delta$.

Want: $n + kT \gg m(\log m + C) + T(\log(d+1) + 1 + C) + n + C$
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$
re-randomizations.

Want: $n + kT \gg m(\log m + C) + T(\log(d+1) + 1 + C) + n + C$
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$
re-randomizations.

Want: $n + kT \gg m \log m + C + T \log (d+1) + 1 + C + n + C$

Aka: $m \log m + C \ll T (k - \log (d+1) + 1 + C)$
Win if random bits in $\gg$ bits out

aka, then we’d get a contradiction and conclude that there must be $< T$ re-randomizations.

Want:

\[ n + kT \gg m(\log m + C) + T(\log(d+1)+1+C) + n + C \]

Aka:

\[ m(\log m + C) \ll T(k - \log(d+1)+1+C) \]

Provided that $k \geq \log(d+1) + 100000$, this happens for $T = \text{poly}(m)$.
What happens if there are $t > 2$ colors?
Our algorithm

- **FindSat**$(S_1, S_2, ..., S_m)$:
  - Choose a random coloring $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**(i, $\sigma$):
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, ..., S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, ..., d + 1$:
    - If $S_{i_j}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
    - Return $\sigma$

Fixing set $i$!

All done with this level.

Trying to fix
the $j$’th child

...because it was all red!
(or blue, or purple, or.... as appropriate)

After $T$ re-randomizations,
I give up. I’ve got $\sigma$
Recovering the random bits

• The print statements allow us to reconstruct the recursion tree.
• Then...

Since I know the recursion tree, I know that at this point “the j’th child” means $S_4$.

Now we know what this assignment was, so we can keep working backwards!

Say we know the coloring AFTER we re-randomized to fix the j’th child. (We know the final assignment since it was printed out, and we’re working backwards.)
To do the proof

- We need to count the number of random bits that go in in the first $T$ re-randomizations.
- We need to count the number of bits of print statements that come out in the first $T$ re-randomizations.
- We need to argue that we can recover the random bits that go in from the print statements that come out.

![Diagram](image)
Our algorithm

- **FindSat**($S_1, S_2, ..., S_m$):
  - Choose a random *coloring* $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**$(i, \sigma)$:
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, ..., S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, ..., d + 1$:
    - If $S_{i_j}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
    - Return $\sigma$

Random bits in:

$$n + \frac{k \cdot T}{T} \cdot \log(t)$$

$k \cdot \log(t)$ bits per re-randomization.
Our algorithm

- **FindSat**($S_1, S_2, ..., S_m$):
  - Choose a random coloring $\sigma$ for each of numbers
  - For each $S_i$ that is monochromatic:
    - $\sigma \leftarrow \text{Fix}(i, \sigma)$
    - Return $\sigma$

- **Fix**($i, \sigma$):
  - Update $\sigma$ by re-randomizing every number in $S_i$
  - Let $S_{i_1}, S_{i_2}, ..., S_{i_{d+1}}$ be the sets that intersect $S_i$
  - For $j = 1, ..., d + 1$:
    - If $S_{i_j}$ is monochromatic:
      - $\sigma \leftarrow \text{Fix}(i_j, \sigma)$
      - Return $\sigma$

Bits out:

- $\leq m[\log(m) + C]
  \begin{align*}
  &= \log(t) \\
  &+ T[\log(d+1) + 1 + C] \\
  &+ n + C
  \end{align*}
  \text{"all done"}

- Also “all done” because it was all red! (or blue, or purple, or... as appropriate)
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$
re-randomizations.

Random bits in:

$$n + k \cdot T \log(t)$$

Bits out:

$$\leq m \left[ \log(m) + C \right] + T \left[ \log(d+1) + 1 + C \right] + n + C$$

- "Fixing clause $i$"
- "Trying to fix $i$th child because it was red"
- "I give up, δ."
- "Call Fix $T$ times"
Win if random bits in $\gg$ bits out

aka, then we’d get a contradiction and conclude that there must be $< T$ re-randomizations.

Random bits in: $n + k \cdot T \log(t)$

Bits out: $\leq m \left[ \log(m) + C \right] + T \left[ \log(d+1) + 1 + C \right] + n + C$

Want: $n + kT \log(t) \gg m(\log m + C) + T(\log(d+1) + 1 + C) + n + C$
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$ re-randomizations.

Want: $n + kT^{\log(t)} \gg m(\log m + C) + T(\log(d+1) + 1 + C) + n + C$
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$
re-randomizations.

Want: \[ n + kT^\log(t) \gg m(\log m + C) + T (\log(d+1) + 1 + C) + n + C \]

Aka: \[ m(\log m + C) \ll T (k - \log(d+1) + 1 + C) \]
Win if random bits in $\gg$ bits out
aka, then we’d get a contradiction and conclude that there must be $< T$ re-randomizations.

Want: $n + kT^{\log(t)} \gg m(\log m + C) + T(\log(d+1) + 1 + C) + n + C$

Aka: $m(\log m + C) \ll T(kG^{\log(t)} - \log(d+1) + 1 + C)$

Provided that $k \geq \frac{\log(d+1) + \log(t)}{\log(t)} + 9999 = \frac{\log(d+1)}{\log(t)} + 10000$

This happens for $T = \text{poly}(m)$
Conclusion

As long as \( k \geq \frac{\log(d+1)}{\log(t)} + 10000 \), we can find a good coloring with \( \text{poly}(m) \) re-randomizations!
How does this compare to the general constructive LLL in the lecture notes?

**Corollary 3.** Let $V$ be a finite set of independent random variables. Let $\mathcal{A}$ be a finite set of events determined by the random variables in $V$. If for all $A \in \mathcal{A}$, $|\Gamma(A)| \leq d+1$, and $\Pr[A] \leq \frac{1}{e(d+1)}$, then Algorithm 2 will find an assignment to the variables $V$ such that no event of $\mathcal{A}$ occurs. Additionally, the expected number of “re-randomizations” performed by the algorithm is bounded by $O(|\mathcal{A}|/(d+1))$. 
How does this compare to the general constructive LLL in the lecture notes?

**Corollary 3.** Let $V$ be a finite set of independent random variables. Let $A$ be a finite set of events determined by the random variables in $V$. If for all $A \in A$, $|\Gamma(A)| \leq d+1$, and $\Pr[A] \leq \frac{1}{e(d+1)}$, then Algorithm 2 will find an assignment to the variables $V$ such that no event of $A$ occurs. Additionally, the expected number of “re-randomizations” performed by the algorithm is bounded by $O(|A|/(d+1))$.

$A_i = \{S_i \text{ is monochromatic}\}$

$\Pr\{A_i\} = \frac{t}{t^k}$ for $t$ colors.
How does this compare to the general constructive LLL in the lecture notes?

**Corollary 3.** Let $V$ be a finite set of independent random variables. Let $A$ be a finite set of events determined by the random variables in $V$. If for all $A \in A$, $|\Gamma(A)| \leq d+1$, and $\Pr[A] \leq \frac{1}{e(d+1)}$, then Algorithm 2 will find an assignment to the variables $V$ such that no event of $A$ occurs. Additionally, the expected number of “re-randomizations” performed by the algorithm is bounded by $O(|A|/(d+1))$.

$A_i = \{S_i \text{ is monochromatic}\}$

$\Pr\{A_i\} = \frac{t}{t^k}$ for $t$ colors.

Need: $\Pr\{A_i\} \leq \frac{1}{e(d+1)}$
How does this compare to the general constructive LLL in the lecture notes?

**Corollary 3.** Let \( V \) be a finite set of independent random variables. Let \( \mathcal{A} \) be a finite set of events determined by the random variables in \( V \). If for all \( A \in \mathcal{A}, |\Gamma(A)| \leq d+1 \), and \( \Pr[A] \leq \frac{1}{e(d+1)} \), then Algorithm 2 will find an assignment to the variables \( V \) such that no event of \( \mathcal{A} \) occurs. Additionally, the expected number of “re-randomizations” performed by the algorithm is bounded by \( O(|\mathcal{A}|/(d+1)) \).

\[ A_i = \left\{ S_i \text{ is monochromatic} \right\} \]

\[ \Pr\{A_i\} = \frac{t}{t^k} \text{ for } t \text{ colors.} \]

Need: \[ \Pr\{A_i\} \leq \frac{1}{e(d+1)} \]

\[ t - (k-1) \leq \frac{1}{e(d+1)} \]

\[ (k-1) \log(t) \geq 1 + \log(d+1) \]
How does this compare to the general constructive LLL in the lecture notes?

**Corollary 3.** Let $V$ be a finite set of independent random variables. Let $A$ be a finite set of events determined by the random variables in $V$. If for all $A \in A$, $|\Gamma(A)| \leq d+1$, and $\Pr[A] \leq \frac{1}{e(d+1)}$, then Algorithm 2 will find an assignment to the variables $V$ such that no event of $A$ occurs. Additionally, the expected number of "re-randomizations" performed by the algorithm is bounded by $O(|A|/(d+1))$.

$A_i = \{S_i \text{ is monochromatic}\}$

$\Pr\{A_i\} = \frac{t}{t^k}$ for $t$ colors.

Need:

\[
\Pr\{A_i\} \leq \frac{1}{e(d+1)}
\]

\[
t - (k-1) \leq \frac{1}{e(d+1)}
\]

\[(k-1) \log(t) \geq 1 + \log(d+1)
\]

\[k \geq \frac{\log(d+1)}{\log(t)} + \text{[constant]}
\]

Same thing!
Conclusions

• As long as \( k \geq \frac{\log(d+1)}{\log(t)} + 10000 \), we can find a good coloring with \( \text{poly}(m) \) re-randomizations!

• You now have some idea of how you might adapt this proof to deal with other examples (aka, ones with \( \Pr[A_i] \leq p \) for a general \( p \))....

• is a very cute idea.

(This method of proof is called “entropy compression”)

Algorithm

Random bits

Print statements

Formula \( \varphi \) → Algorithm → Assignment \( \sigma \)