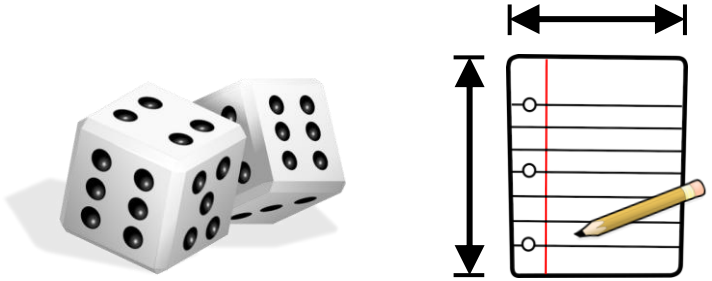
OpenAI's Proof That $\mathbf{L} = \mathbf{BPL}$

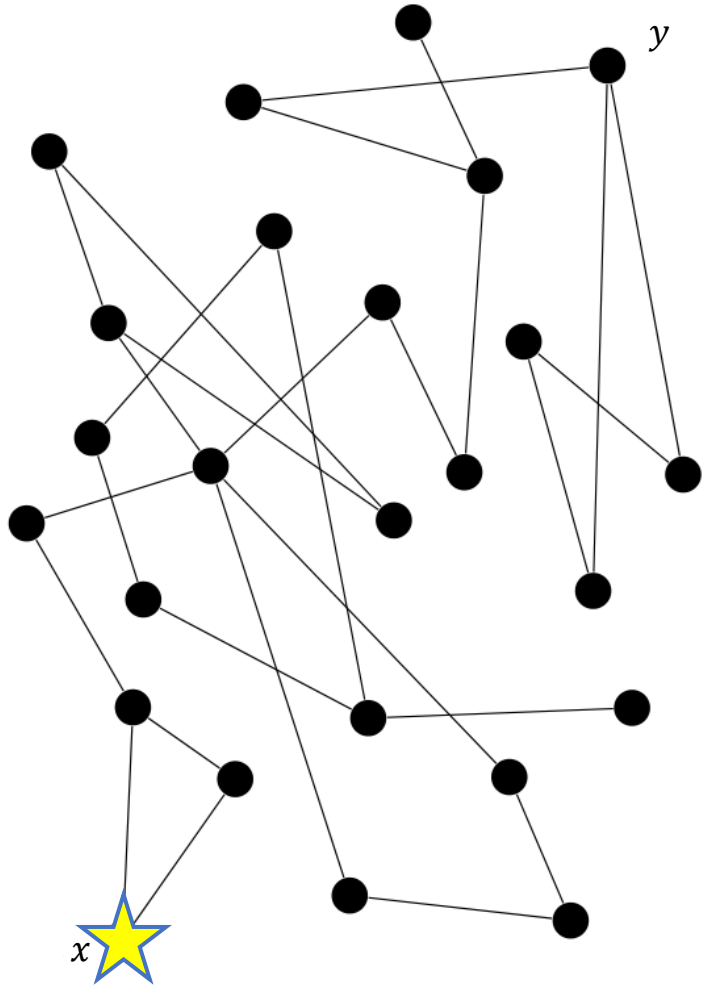


William Hoza

The University of Chicago

November 17, 2026

A motivating example



- **Given:** Undirected graph G and vertices x, y

Theorem [Aleliunas, Karp, Lipton, Lovász, Rackoff 1979]:

∃ randomized algorithm that determines whether there is a path from x to y using $O(\log n)$ bits of space

- **Algorithm**: Take a random walk from x of length $\mathrm{poly}(n)$ and check whether we visit y. (This works!)



No known efficient algorithm



Efficient randomized algorithm



Efficient deterministic algorithm



Derandomization

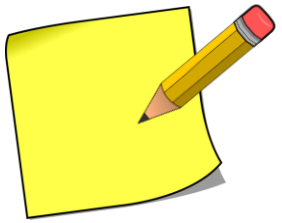


- **Theorem** [Reingold 2008]: There is a deterministic log-space algorithm for undirected connectivity
- **Theorem** [OpenAI 2026]: Every problem that can be solved by a randomized log-space algorithm can also be solved by a deterministic log-space algorithm!

The complexity class BPL



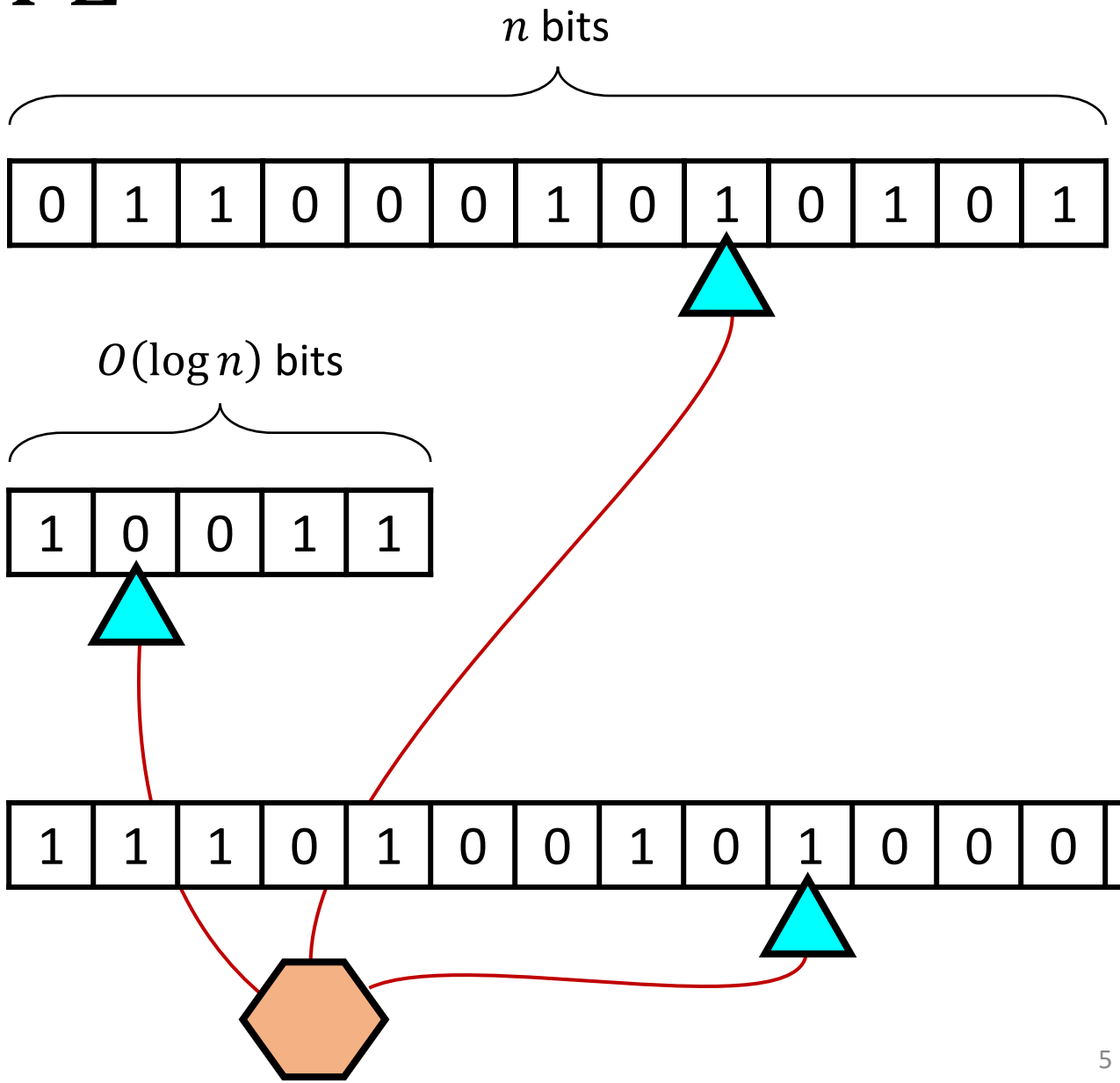
Read-only input tape



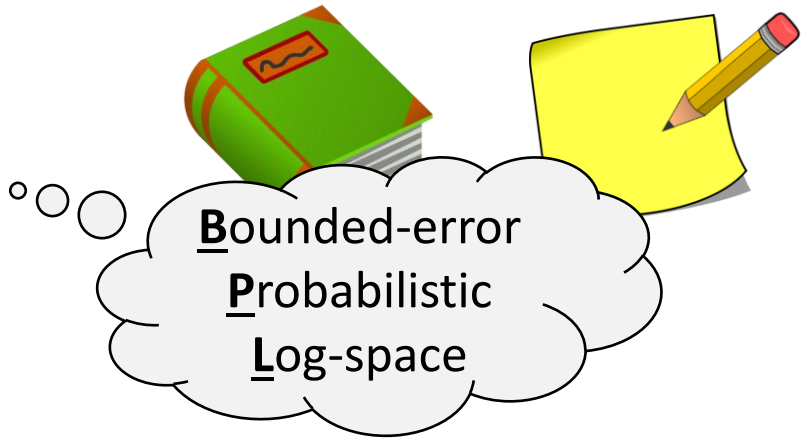
Read-write work tape



Read-once random tape



The complexity class BPL





- Let $L: \{0,1\}^* \to \{0,1\}$
- **Definition:** $L \in$ BPL if there exists a Turing machine M such that:
  - There is a read-only tape storing the n-bit input
  - There is a read/write work tape of size $O(\log n)$
  - There is a read-once tape filled with random bits
  - For every $w \in \{0,1\}^*$, we have $\Pr[M(w) = L(w)] \geq 2/3$
  - M halts for every input and every setting of the random tape

Exercise: Show that M halts in polynomial time

Derandomization of BPL



- Trivially, $\mathrm{L} \subseteq \mathrm{BPL}$

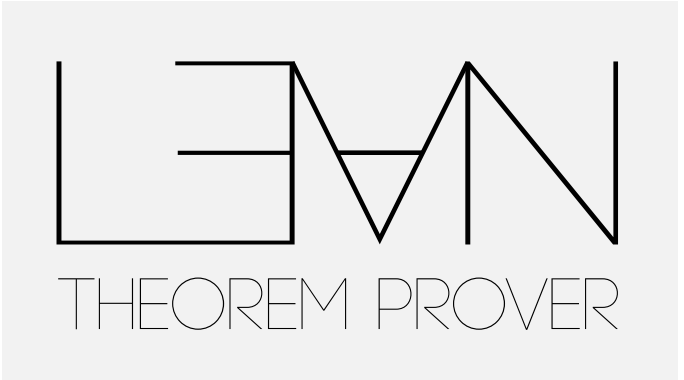
Theorem [OpenAI 2026]: $\mathrm{L} = \mathrm{BPL}$

More general theorem [OpenAI 2026]: $\forall S \geq \log n$, we have

$$\mathrm{BPSPACE}\big(O(S)\big) = \mathrm{DSPACE}\big(O(S)\big)$$

- "Randomness is not necessary for space-efficient computation!"

Correctness of the proof



- The proof is AI-generated
- There is a Lean-verified version, so the proof is almost certainly correct ✅
- That's nice, but we were already confident that $\mathrm{L} = \mathrm{BPL}$
- We want to know why the theorem is true

Disclaimers

- The proof is long and complicated, and I don't fully understand it yet!
- **Open problem:** Simplify the proof
- I'll sketch a weaker $\tilde{O}(\log n)$ space proof…
- …which I *also* don't fully understand!

Disclaimers

- My (incomplete) understanding of the proof is mostly based on discussions with ChatGPT 🤞
- Thanks also to many wonderful humans for helpful discussions
  - Zander Kelley, Ian Mertz, Louie Putterman, Avishay Tal, Salil Vadhan, …
- Mistakes are my own
- Some false things I'll say are deliberate simplifications

The concrete problem we will solve

- **Given:** N-vertex directed acyclic graph G, start vertex x, accept vertex y
- **Goal:** Estimate the probability that a random walk from x visits y, to within $\pm \frac{1}{N}$

Theorem [OpenAI 2026]: There is a deterministic algorithm that solves this problem using $O(\log N)$ bits of space

- **Exercise:** The theorem implies $\mathrm{L} = \mathrm{BPL}$

Laplacian approach

"Inverse Laplacian perspective" used by many works on L vs. BPL, starting with [Murtagh, Reingold, Sidford, and Vadhan, FOCS 2017]

- Let $C_0 \in [0,1]^{N \times N}$ be the random walk matrix of G
  - $C_0(x, y) =$ probability of going from x to y in one step
- **Goal:** Approximate the matrix $C_0^0 + C_0^1 + \cdots + C_0^N = (I - C_0)^{-1}$
- **Strategy (v1):** Compute matrices $E_1, E_2, \ldots, E_L$ where $L = O(\log N)$ such that

$$(I + E_L) \cdots (I + E_2) \cdot (I + E_1) \cdot (I - C_0) \approx I$$

- Then $(I - C_0)^{-1} \approx (I + E_L) \cdot (I + E_{L-1}) \cdots (I + E_1)$

Interpretation: Shortcutting

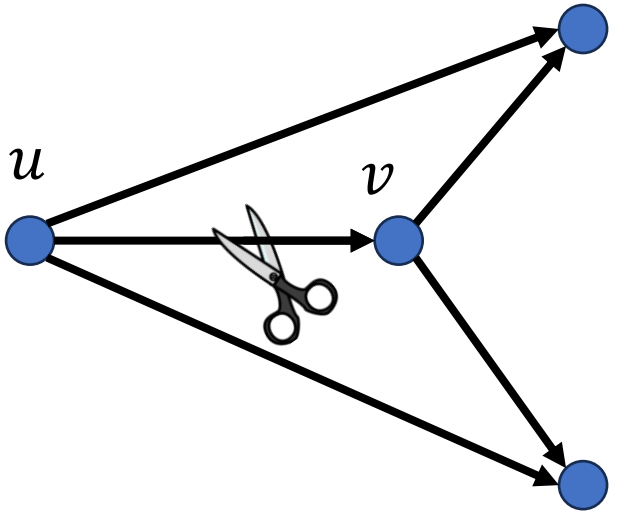
But it will re-appear going from u to out-neighbors of v

$$(I + E) \cdot (I - C) = I - C' \text{ where } C' = C - E + EC$$

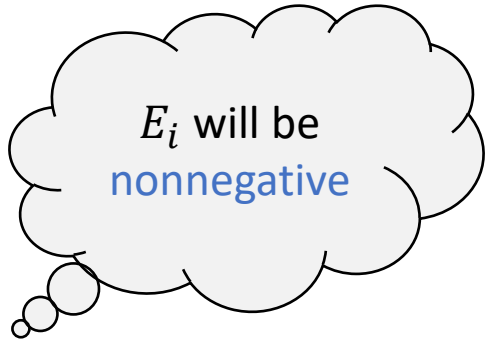
We can remove weight from an edge (u, v)…



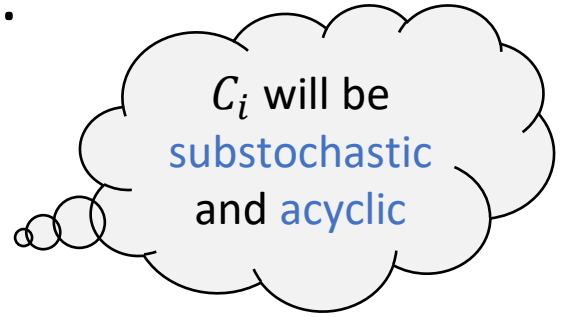
Photo by Wolfgang Sauber. License: CC-BY-SA 3.0



Recap of strategy



- **Goal**: Construct matrices $E_1, E_2, \ldots, E_L$ such that:
  - $L = O(\log N)$
  - $C_L \approx 0$, where $C_i = C_{i-1} - E_i + E_i C_{i-1}$
  - We can approximately compute E_i using $O(\log N)$ bits of space

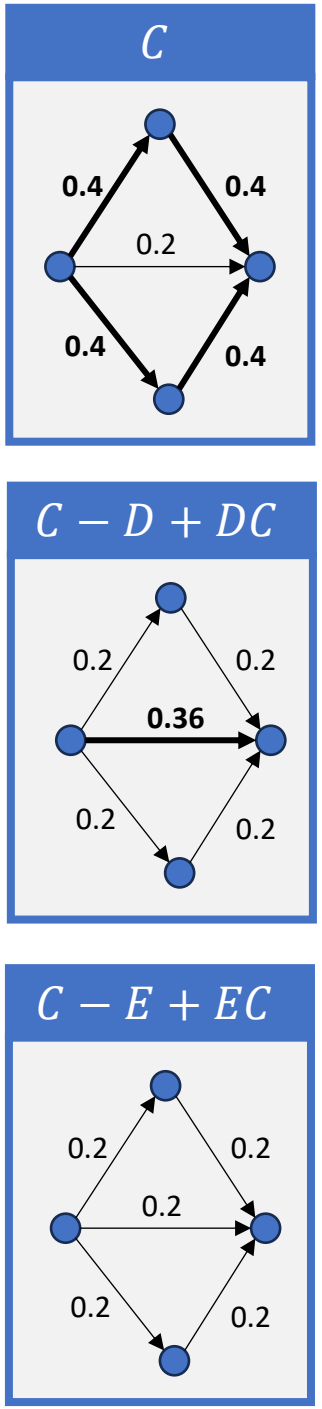


- Overhead for multiplying $(I + E_L) \cdots (I + E_2) \cdot (I + E_1)$ will only be $O(\log N \cdot \log\log N)$ bits of space ✅

Progress measure: Maximum edge weight

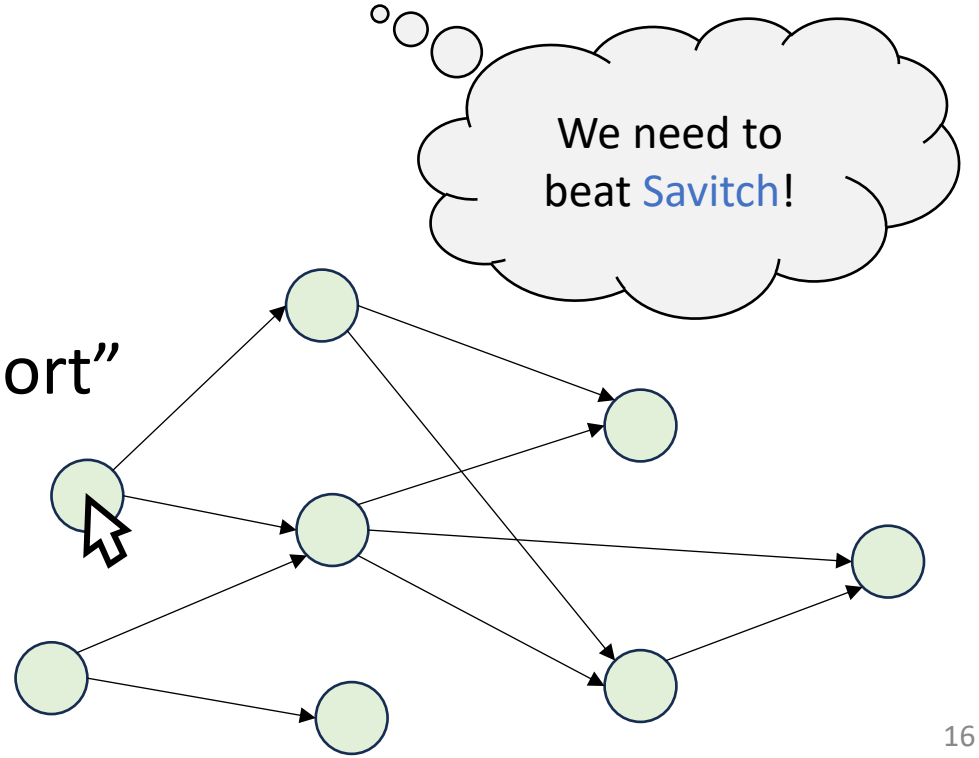
- Let $f = \|C\|_{\max}$
- **Chop #1:** Let $D(x,y) = (C(x,y) - f/2)_+$
- **Chop #2:** Let $E = D + (C - D + DC - f/2)_+$
- **Lemma:** $\|C - E + EC\|_{\max} \leq 5f/6$. (Proof omitted)
- So $\|C_L\|_{\max} \leq 1/\mathrm{poly}(n)$ after $L = O(\log N)$ iterations ✅

Notation:
$e_+ := \max(e, 0)$
$= \mathrm{ReLU}(e)$



Main challenge: Space complexity

- **Goal:** Approximately compute E_i using only $O(\log N)$ bits of space
- At each moment, just one vertex stored in memory across all recursion levels!
  - Global "cursor"
- Describe an edge using one endpoint and a "port"
  - "p-th outgoing / incoming edge at x"
  - Think: Adjacency list

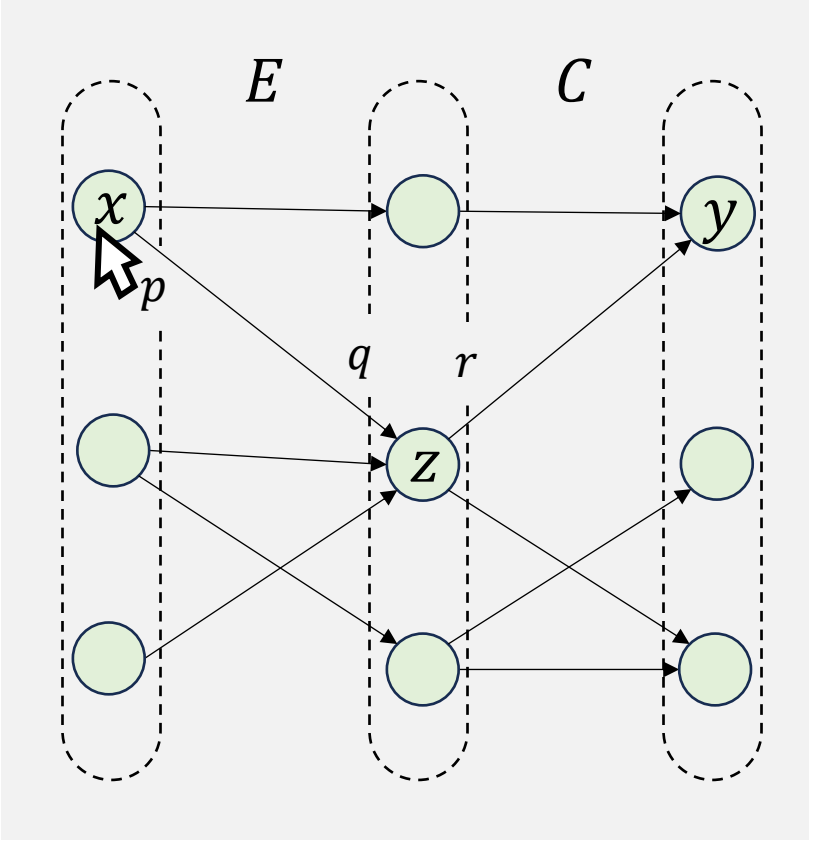


Space complexity

- $C' = C - E + EC$
- Say we are trying to compute $E(x, z) \cdot C(z, y)$

Time

Cursor	Additional memory			
x	p	r		
x	p	r	$E(x,z)$	
z	q	r	$E(x,z)$	
z	q	r	$E(x,z)$	$C(z,y)$
x	p	r	$E(x,z)$	$C(z,y)$



Wish:

- E has degree $O(1)$
- C has out-degree $O(1)$

Example: Out-degree of E

Notation: $\deg^+(E)$ means maximum unweighted out-degree

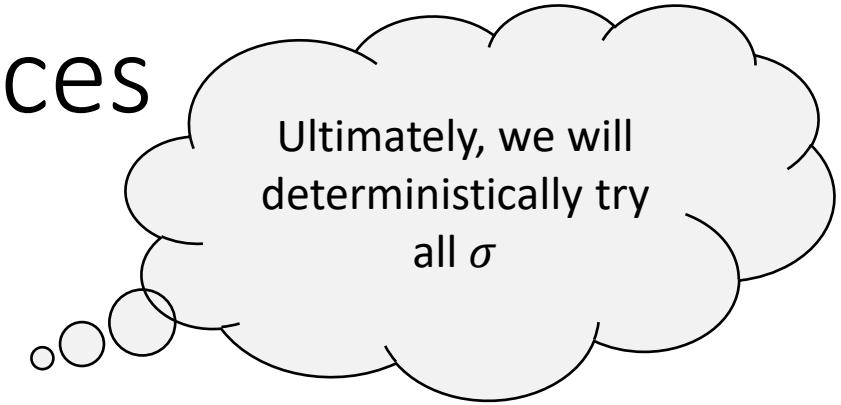
- $f = \|C\|_{\max}$
- $E(x,y) = (C(x,y) - f/2)_+$ *[ignoring chop #2]*
- **Wish:** $\deg^+(E) = O(1)$
- **Reality:** $\deg^+(E) = O(1/f)$ and might be $\Theta(1/f)$

Sparsification

Edges vs. vertices

- Conventional approach (e.g., [Nisan 1992]):
  - Pick a pseudorandom subset of outgoing ports $S \subseteq [D]$
  - Keep p-th outgoing edge of x for each $p \in S$
- New approach:
  - Pick pseudorandom subsets of vertices $X, Y \subseteq [N]$
  - Keep edges in $X \times Y$

Pseudorandom subsets of vertices



- Let $\sigma \in \{0,1\}^{O(\log n)}$ be a random seed
- *(Simplified)* Pick a random hash function $T: [N] \to [0,1]$ such that $T(1), T(2), \ldots, T(N)$ are uniform and pairwise independent
- Let U, V be independent copies of T
- For $X \in \{T, U, V\}$, let $X_u = \{x \in [N] : X(x) \leq u\}$, so $\Pr[x \in X_u] = u$
- **Idea:** Only keep edges in, e.g., $T_f \times U_f$

"channel" in OpenAI manuscript

Sparsified graphs

- **Definition (v1):** For $u, t \in [0,1]$ and distinct $X, Y \in \{T, U, V\}$, let

$$C^{X_u, Y_t}(x, y; \sigma) = C(x, y) \cdot 1[x \in X_u] \cdot 1[y \in Y_t]$$

$$E^{X_u, Y_t}(x, y; \sigma) = E(x, y) \cdot 1[x \in X_u] \cdot 1[y \in Y_t]$$

- For now, let's optimistically assume C^{X_f, Y_f} and E^{X_f, Y_f} have degree $O(1)$
  - Not literally true – we will revisit this optimistic assumption later if we have time

Computing the sparsified matrices

- Assume we can compute $E^{X,Y}$ and $C^{X,Y}$
- How do we compute $(C')^{X,Y}$?
- Recall $C' = C - E + EC$
- **Note:** $(EC)^{X,Y} \neq E^{X,Y} C^{X,Y}$

A simplifying assumption

- Recall $f = \|C\|_{\max}$
- For simplicity, assume every nonzero entry of C is $\Theta(f)$
  - Also assume the same about C'
- (Actual proof: Divide into bins, treat each bin separately, etc.)

Approximating the sparsified graphs

- **Induction hypothesis:** $\forall\, u, t \le f$, $\forall$ distinct $X, Y \in \{T, U, V\}$, can compute sparse graphs $\hat{C}^{X_u, Y_t}, \hat{E}^{X_u, Y_t}$ such that whp over σ, we have

"estimated tables" in OpenAI manuscript

$$\begin{cases} \hat{C}^{X_u, Y_t}(x, y; \sigma) \approx C^{X_u, Y_t}(x, y; \sigma) \\ \hat{E}^{X_u, Y_t}(x, y; \sigma) \approx E^{X_u, Y_t}(x, y; \sigma) \end{cases}$$

"true tables" in OpenAI manuscript

- Can achieve error 2^{-m} using:
  - $O(m + \log(1/u))$ bits of work space

$+O(\log n)$ bits of scratch space, erased before each recursive call

Also, we're ignoring $\log\log N$ terms

  - Global vertex register x ("cursor") and seed register σ, both treated as catalytic memory

Approximating $(EC)^{X,Y}$

Three channels X, Y, Z with shifting roles, cf. [Ben-Or and Cleve 1992]

- $C' = C - E + EC$
- Focus on EC
- **Goal:** Approximate $(EC)^{X_u,Y_t}(x, y; \sigma) \coloneqq (EC)(x, y) \cdot 1[x \in X_u, y \in Y_t]$
- **Step 1:** Let $\widetilde{EC}^{X_u,Y_t}(x, y; \sigma) = \frac{1}{f} \sum_z \hat{E}^{X_u,Z_f}(x, z; \sigma) \cdot \hat{C}^{Z_f,Y_t}(z, y; \sigma)$

"raw sample term" in OpenAI manuscript

$$\widetilde{EC}^{X_u,Y_t}(x,y;\sigma) = \frac{1}{f}\sum_z \hat{E}^{X_u,Z_f}(x,z;\sigma)\cdot\hat{C}^{Z_f,Y_t}(z,y;\sigma)$$

$$\overset{\text{whp}}{\approx} \frac{1}{f}\sum_z E(x,z)\cdot C(z,y)\cdot 1[x\in X_u]\cdot 1[y\in Y_t]\cdot 1\left[z\in Z_f\right]$$

- Therefore, $\mathbb{E}_\sigma\left[\,\widetilde{EC}^{X_u,Y_t}(x,y;\sigma)\mid x\in X_u, y\in Y_t\,\right]\approx (EC)(x,y)$ ✅
- **What we have:** Approximation in expectation
- **What we need:** Approximation whp

Approximation whp

- Let $\Omega = \{\sigma : x \in X_u, y \in Y_t\}$
- **Attempt 1:** Let $\widehat{EC}^{X_u,Y_t}(x,y,\sigma) = 1[\sigma \in \Omega] \cdot \frac{1}{|\Omega|} \sum_{\tau \in \Omega} \left[\widetilde{EC}^{X_u,Y_t}(x,y,\tau)\right]$
- **Issue:** Summing over Ω takes too much space
- We only have one global seed register!

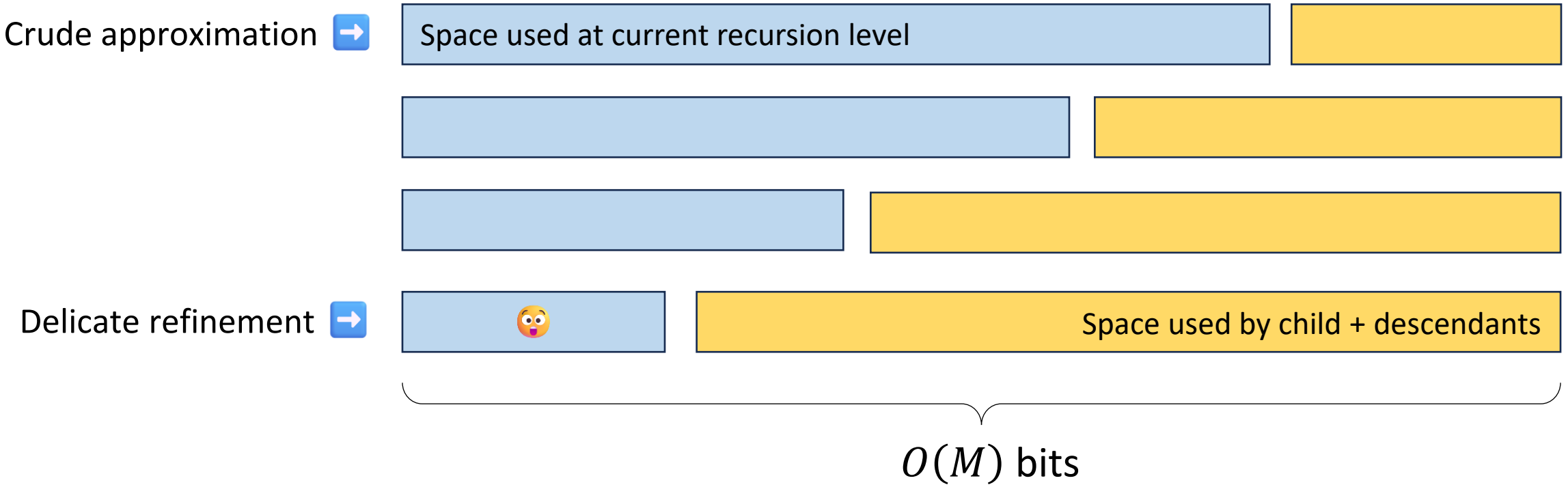


Iterative refinement

Iterative refinement

Similar spirit as:
- Richardson iteration, e.g., [MRSV 2017]
- WPRGs, e.g., [Braverman, Cohen, Garg 2018]
- $\mathrm{BPL} \subseteq$ uniform AC^1 [Cheng, Wang 2024]

- **Plan:** Start with a crude approximation and iteratively refine until error is 2^{-M}



Expander notation

- Let H be a constant-degree expander on $\Omega = \{\sigma : x \in X_u, y \in Y_t\}$
- For a function $F(x, y; \sigma)$, define

$$(HF)(x, y; \sigma) = \frac{1}{\deg(H)} \sum_{(\sigma,\tau)\in E_H} F(x, y; \tau)$$

Iterative refinement

Approximation of $C^{Z,Y}$ with error 2^{-m}

- Let $\widetilde{EC}^{X,Y,m}(x,y;\sigma) = \frac{1}{f}\sum_z \hat{E}^{X,Z,m}(x,z;\sigma) \cdot \hat{C}^{Z,Y,m}(z,y;\sigma)$
- For target error 2^{-M}, define radius sequence $(R_1, R_2, R_3, \dots) = \left(M, \frac{M}{2}, \frac{M}{4}, \dots\right)$
- Let $\widehat{EC}^{X,Y,M} = \sum_{j=1}^{\log M} H^{R_j}\left(\widetilde{EC}^{X,Y,M-R_{j+1}} - \widetilde{EC}^{X,Y,M-R_j}\right)$ where $\widetilde{EC}^{X,Y,0} \equiv 0$

Lemma: Let $F, F_1, \dots, F_M : \Omega \to [0,1]$. Assume $\|F - F_m\|_2 \le 2^{-m}$.

Then $\left\|\mathbb{E}[F] - \sum_{j=1}^{\log M} H^{R_j}\left(F_{M-R_{j+1}} - F_{M-R_j}\right)\right\|_2 \le O(2^{-M})$.

Proof: Exercise

Space complexity

$$\widehat{EC}^{X,Y,M} = \sum_{j=1}^{\log M} H^{R_j} \left(\widetilde{EC}^{X,Y,M-R_{j+1}} - \widetilde{EC}^{X,Y,M-R_j} \right)$$

- Heuristically, space complexity of term j should be

$$\underbrace{O(R_j)}_{\text{Expander walk}} + \underbrace{O(M - R_j)}_{\text{Recursion}} = O(M)$$ 🙂

Sparsity

- **What we need:** A sparse, space-efficient approximation of $(C')^{X,Y}$
- **What we've constructed:** An approximation of the form $\sum_{j=1}^{\log M} \Delta_j$
- Update Δ_j has the form $\Delta_j = H^{R_j} \cdot \left(\tilde{F}_{M-R_{j+1}} - \tilde{F}_{M-R_j}\right)$, where $\tilde{F}_m$ combines $\hat{C}^{X,Y,m}$, $\hat{E}^{X,Y,m}$, $\hat{C}^{Z,Y,m}$, $\hat{E}^{X,Z,m}$
- With each update, approximation gets more accurate 🙂
- But it also gets more dense 😟

Compression

- Dense approximation: $(C')^{X,Y} \approx \sum_{j=1}^{\log M} \Delta_j$
- Sparse approximation: Let $T_0 = 0$ and $T_j = \text{Compress}_j\left(T_{j-1} + \Delta_j\right)$
  - $\text{Compress}_j(T)$ deletes edges with absolute weight less than $f \cdot 2^{-cR_j}$
- Now the approximations $T_1, \ldots, T_{\log M}$ are getting sparser and sparser!
- $\deg^+\left(T_j\right) = 2^{O\left(R_j\right)}$ culminating in $\deg^+\left(T_{\log M}\right) = O(1)$!

Error from compression

$$\widehat{C'}^{X,Y,M} = T_{\log M} \text{ where } T_0 = 0 \text{ and } T_j = \text{Compress}_j\left(T_{j-1} + \Delta_j\right)$$

- **Claim:** Compress_j doesn't introduce much approximation error
- **Intuition:**
  - **Case 1:** j is small (crude approximation). Then Compress_j only removes edges that have near-zero weight ✅
  - **Case 2:** j is large (delicate refinement). By simplifying assumption, every nonzero edge weight in $(C')^{X,Y}$ is $\Theta(f)$, so Compress_j only removes erroneous edges ✅

Space complexity, revisited

$$\widehat{C'}^{X,Y,M} = T_{\log M} \text{ where } T_0 = 0 \text{ and } T_j = \text{Compress}_j\left(T_{j-1} + \Delta_j\right)$$

- **So far:** Adjacency matrix of $\widehat{C'}^{X,Y}$ in terms of adjacency matrices of $\hat{C}^{X,Y}$, $\hat{E}^{X,Y}$
- **Need:** Adjacency list of $\widehat{C'}^{X,Y}$ in terms of adjacency lists of $\hat{C}^{X,Y}$, $\hat{E}^{X,Y}$!
- How do we compute information about the p-th outgoing edge of x?
- ⚠️ p-th outgoing edges in C and C' may lead to different destinations!

Adjacency lists

Space complexity of adjacency list

$$\widehat{C'}^{X,Y,M} = T_{\log M} \text{ where } T_0 = 0 \text{ and } T_j = \text{Compress}_j\left(T_{j-1} + \Delta_j\right)$$

Lemma: Can compute information about p-th outgoing edge of x in T_j using $O\left(R_j\right)$ bits of space, plus queries to T_{j-1}, $\tilde{F}_{M-R_j}$, and $\tilde{F}_{M-R_{j+1}}$,

plus global vertex/seed registers, plus log log terms that we ignore, plus $O(\log n)$ bits of scratch space that is erased before any queries

- As we will see, the proof of the lemma isn't hard – just unpack definitions

Merging edges

- $T_j = \text{Compress}_j(T_{j-1} + \Delta_j)$
- Let's be a little more precise about the notation:
  - $T_{j-1} + \Delta_j$ denotes multi-graph constructed by interpreting $+$ as union of edge sets
  - $\text{Merge}(T_{j-1} + \Delta_j)$ denotes graph in which we merge edges with same endpoints
  - $\text{Compress}_j(T_{j-1} + \Delta_j)$ deletes light-weight edges in $\text{Merge}(T_{j-1} + \Delta_j)$

Navigating T_j

- $T_j = \text{Compress}_j(T_{j-1} + \Delta_j)$
- To compute information about the p-th outgoing edge of x in T_j:
  1. Initialize $r \leftarrow 0$
  2. For $q \in \left[2^{O(R_j)}\right]$:
     a) If absolute weight of q-th outgoing edge of x in $\text{Merge}(T_{j-1} + \Delta_j)$ is $\geq f \cdot 2^{-cR_j}$, increment r
     b) If $r = p$, then we've found the edge we're interested in
- Space $= O(R_j) +$ space complexity of navigating $\text{Merge}(T_{j-1} + \Delta_j)$

Navigating $\mathrm{Merge}(T_{j-1} + \Delta_j)$

- Let Q be the set of q such that p-th and q-th outgoing edges of x in $T_{j-1} + \Delta_j$ have the same destination
- If $p \neq \min(Q)$, then p-th outgoing edge of x in $\mathrm{Merge}(T_{j-1} + \Delta_j)$ is `null`
- If $p = \min(Q)$, then weight of edge in $\mathrm{Merge}(T_{j-1} + \Delta_j)$ is sum of weights of Q-edges in $T_{j-1} + \Delta_j$
- Space $= O(R_j)$ + space complexity of navigating $T_{j-1} + \Delta_j$, including comparing destinations

Comparing destinations

- To figure out whether p-th and q-th outgoing edges of x lead to the same destination in $T_{j-1} + \Delta_j$:
  1. For $k \in [\log N]$:
     a) Compute k-th bit of name of destination of p-th outgoing edge of x in $T_{j-1} + \Delta_j$
     b) Compute k-th bit of name of destination of q-th outgoing edge of x in $T_{j-1} + \Delta_j$
     c) If they don't match, output "No"
  2. Output "Yes"
- Space $= O(\log \log N) +$ space complexity of computing destinations in $T_{j-1} + \Delta_j$

Navigating $T_{j-1} + \Delta_j$

- $\Delta_j = H^{R_j} \cdot \left(\tilde{F}_{M-R_{j+1}} - \tilde{F}_{M-R_j}\right)$, a union of $2^{O(R_j)}$ graphs
- To compute information about the p-th outgoing edge of x in $T_{j-1} + \Delta_j$, figure out which underlying graph we care about and query it
- That completes the proof of:

Lemma: Can navigate T_j using $O(R_j)$ bits of space, plus queries to T_{j-1}, $\tilde{F}_{M-R_j}$, and $\tilde{F}_{M-R_{j+1}}$, plus unimportant terms

Adding up space used for all T_j

Lemma: Can navigate T_j using $O(R_j)$ bits of space, plus queries to T_{j-1}, $\tilde{F}_{M-R_j}$, and $\tilde{F}_{M-R_{j+1}}$, plus unimportant terms

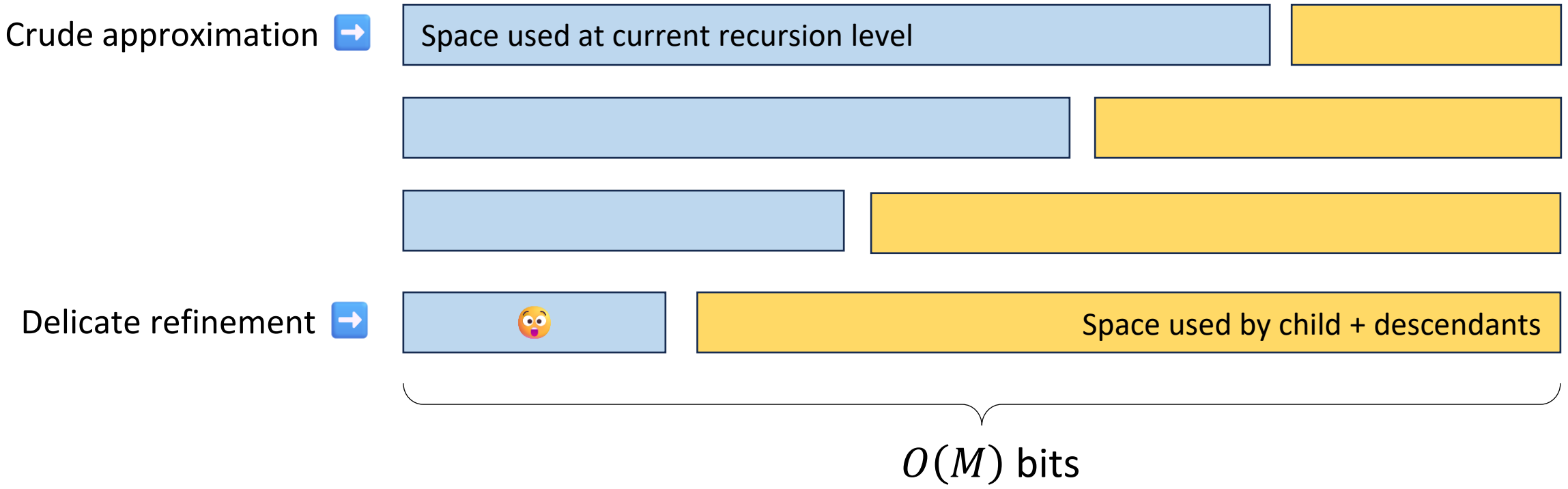
Query	Space used
$T_{\log M}$	$c_1 \cdot R_{\log M}$
$T_{\log M-1}$	$c_1 \cdot R_{\log M-1}$
⋮	⋮
T_j	$c_1 \cdot R_j$
$\tilde{F}_{M-R_{j+1}}$	$c_2 \cdot (M - R_{j+1})$

$$\text{Total} = c_1 \cdot \left(1 + 2 + 4 + \cdots + M/2^{j-1}\right) + c_2 \cdot \left(M - M/2^j\right)$$

$$< 4c_1 \cdot M/2^j + c_2 \cdot \left(M - M/2^j\right)$$

$$\leq c_2 \cdot M, \text{provided we choose } c_2 \geq 4c_1$$

Iterative refinement



Implementing the expander walk

Conditional expander walk

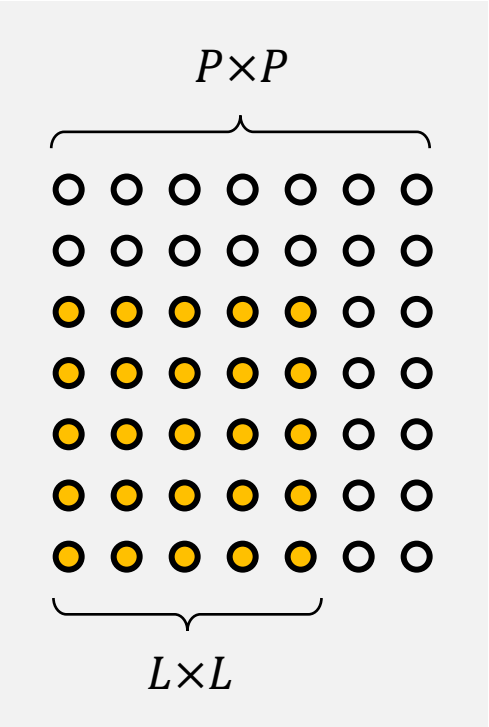
- Recall: To compute information about the p-th outgoing edge of x, we use an expander on the vertex set $\Omega = \{\sigma : x \in X_u, y \in Y_t\}$
- **Difficulty:** We don't know y yet! How can we compute neighbors of σ?
- **Strategy:** Guess and check
- To bound the amount of guesswork, we use a specific, algebraic implementation of the channels X_u, Y_t and the expander graph H

Channel implementation

- Let P be a large enough $O(\log N)$-bit prime and identify $[N] \subseteq \mathbb{F}_P$
- Seed σ specifies $\vec{a}, \vec{b} \in \mathbb{F}_P^2$ where $\vec{a} \neq 0$
- To keep vertices with probability t, keep vertices in the set

$$Y_t \coloneqq \left\{ y \in \mathbb{F}_P : \vec{a}y + \vec{b} \in \mathbb{Z}_L^2 \right\}$$

where $L = P \cdot \sqrt{t}$



Expander construction

- Recall classic MGG expander [Margulis 1973] [Gabber, Galil 1981]
  - Vertex set is $\mathbb{Z}_N \times \mathbb{Z}_N$
  - (a, b) is adjacent to $(a \pm 1, b)$, $(a, b \pm 1)$, $(a \pm b, b)$, $(a, b \pm a)$
- We will do something similar

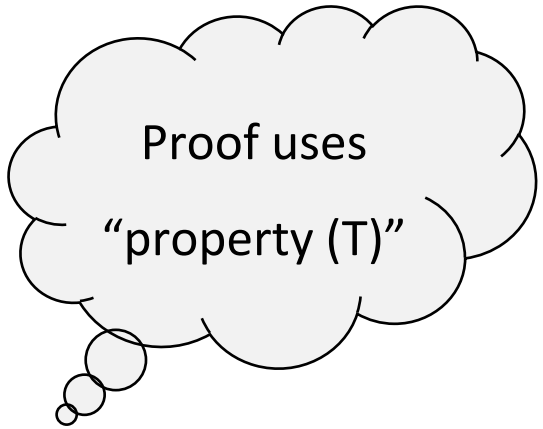
Expander construction

- For simplicity, let's focus on condition $y \in Y_t$, ignore condition $x \in X_u$
- Expander vertex set is $\Omega = \left\{\left(\vec{a}, \vec{b}\right) \in (\mathbb{F}_P^2 \setminus \{0\}) \times \mathbb{F}_P^2 : \vec{a}y + \vec{b} \in \mathbb{Z}_L^2\right\}$
- Expander is easier to describe under a change of coordinates:

$$\left(\vec{a}, \vec{b}\right) \leftrightarrow \left(\vec{a}, \vec{a}y + \vec{b}\right) = (\vec{a}, \vec{c}) \in (\mathbb{F}_P^2 \setminus \{0\}) \times \mathbb{Z}_L^2$$

Expander construction in new coordinates

- Vertex $(\vec{a}, \vec{c}) \in (\mathbb{F}_P^2 \setminus \{0\}) \times \mathbb{Z}_L^2$ has 8 self-loops and 8 other neighbors:
  - $(a_1 + a_2, a_2, c_1 + c_2, c_2)$
  - $(a_1 - a_2, a_2, c_1 - c_2, c_2)$
  - $(a_1, a_2 + a_1, c_1, c_2 + c_1)$
  - $(a_1, a_2 - a_1, c_1, c_2 - c_1)$
  - $(a_1, a_2, c_1 + 1, c_2)$
  - $(a_1, a_2, c_1 - 1, c_2)$
  - $(a_1, a_2, c_1, c_2 + 1)$
  - $(a_1, a_2, c_1, c_2 - 1)$



Theorem: This is a spectral expander, with $\lambda \leq 1 - \Omega(1)$

Expander construction in original coordinates

- $\Omega = \left\{\left(\vec{a}, \vec{b}\right) \in (\mathbb{F}_P^2 \setminus \{0\}) \times \mathbb{F}_P^2 : \vec{a}y + \vec{b} \in \mathbb{Z}_L^2\right\}$ where $y \in \mathbb{F}_P$ is unknown
- Vertex $\left(\vec{a}, \vec{b}\right) \in \Omega$ has 8 self-loops and 8 other neighbors:
  - $(a_1 + a_2, a_2, b_1 + b_2 - o_1 L, b_2)$
  - $(a_1 - a_2, a_2, b_1 - b_2 + o_2 L, b_2)$
  - $(a_1, a_2 + a_1, b_1, b_2 + b_1 - o_3 L)$
  - $(a_1, a_2 - a_1, b_1, b_2 - b_1 + o_4 L)$
  - $(a_1, a_2, b_1 + 1 - o_5 L, b_2)$
  - $(a_1, a_2, b_1 - 1 + o_6 L, b_2)$
  - $(a_1, a_2, b_1, b_2 + 1 - o_7 L)$
  - $(a_1, a_2, b_1, b_2 - 1 + o_8 L)$

where $o_1, \ldots, o_8 \in \{0, 1\}$ are "overflow bits" that depend on y

Guess and check

- We want to take a step in the expander $H_{x,y}$
- We don't know y, so we don't know the relevant overflow bit o_i 😟
- **Strategy:** Guess the overflow bit, then check after we learn y

Guess and check

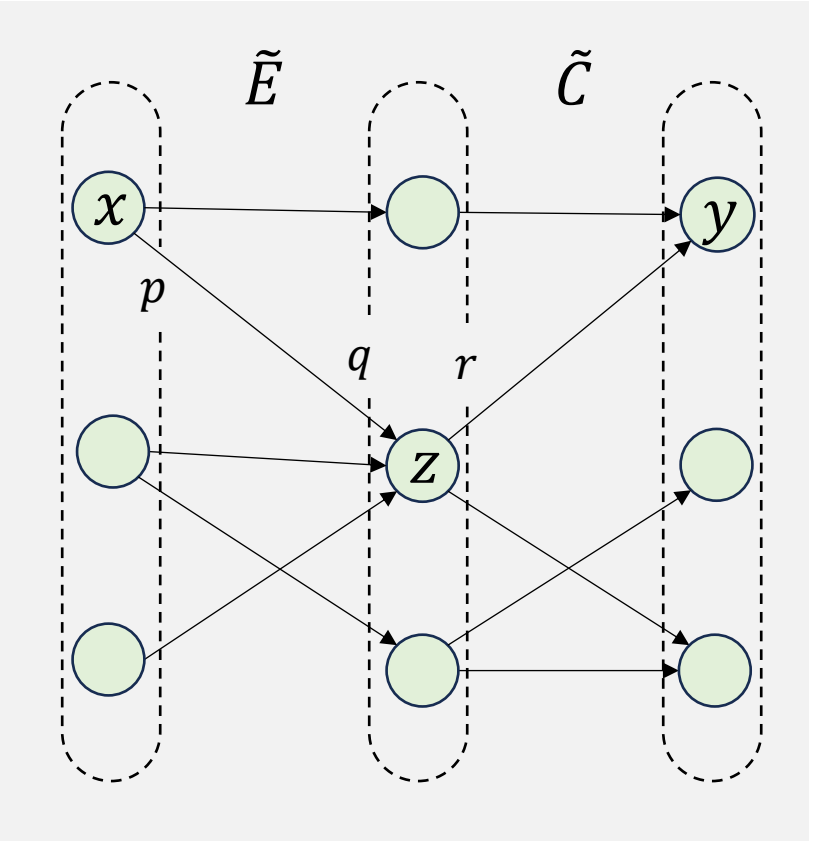
- Suppose we want information about the p-th outgoing edge of x in the multigraph K where $K(x, y; \sigma) = \mathbb{E}_{(\sigma,\tau)\in E\left(H^d_{x,y}\right)} \sum_z \hat{E}^{X,Z}(x, z; \tau)\hat{C}^{Z,Y}(z, y; \tau)$
- Use p to encode d outgoing ports in H, d overflow bit guesses, an outgoing port in $\hat{E}^{X,Z}$, and an outgoing port in $\hat{C}^{Z,Y}$
- Compute destination y, check if $y \notin Y_t$, and check overflow bits
- If $y \notin Y_t$, or if the overflow bits were not guessed correctly, output `null`

Computing $\hat{E}$

- **So far:** How to compute $\hat{C}^{X,Y}$
- **Still need:** How to compute $\hat{E}^{X,Y}$
- **Main additional complication:** Incoming ports and two-way navigation

Two-way navigation

- Recall strategy for computing $\tilde{E}(x,z) \cdot \tilde{C}(z,y)$



Time

Cursor	Additional memory			
x	p	r		
x	p	r	$\tilde{E}(x,z)$	
z	q	r	$\tilde{E}(x,z)$	
z	q	r	$\tilde{E}(x,z)$	$\tilde{C}(z,y)$
x	p	r	$\tilde{E}(x,z)$	$\tilde{C}(z,y)$

- **So far:** How to compute $(x,p) \mapsto z$. **Need to compute:** $(x,p) \leftrightarrow (z,q)$

Incoming ports and two-way navigation

One-way navigator ⇒ two-way navigator

- Generic "upgrade" theorem based on classic work on space-bounded computation [Sipser 1980] [Cook, McKenzie 1987] [Lange, McKenzie, Tapp 2000]
- Roughly: Whenever the map $(x, p) \mapsto z$ is efficiently computable, the maps $(x, p) \leftrightarrow (z, q)$ are also efficiently computable
- Let's prove a clean, simplified version of the theorem

Numberings and rotation maps

These concepts are used in prior work on L vs. BPL, e.g., [Reingold 2008]

- Let A be a simple D-regular directed graph on vertex set V



- A one-way numbering of A is a function $\Gamma: V \times [D] \to V$ such that for every $x \in V$, the vertices $\Gamma(x, 1), \ldots, \Gamma(x, D)$ are the D out-neighbors of x
- A two-way numbering of A is a pair (Γ, Γ'), where Γ is a one-way numbering of A and Γ' is a one-way numbering of "reverse" graph $A^{\top}$
  - If $\Gamma(x, p) = y$ and $\Gamma'(y, q) = x$, we write $\mathrm{Rot}(x, p) = (y, q)$
  - Rotation map $\mathrm{Rot}: V \times [D] \to V \times [D]$



One-way navigator ⇒ two-way navigator

[Sipser 1980] [Cook, McKenzie 1987] [Lange, McKenzie, Tapp 2000]

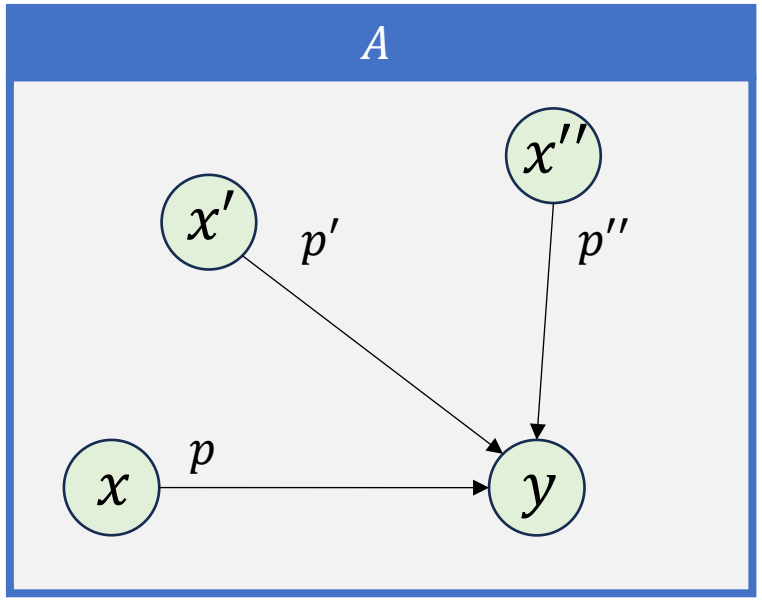
- Assume A has a one-way numbering $\Gamma: V\times[D] \to V$ that can be computed in-place using s bits of scratch space: $(x, p, 0^s) \mapsto (y, 1, 1^s)$

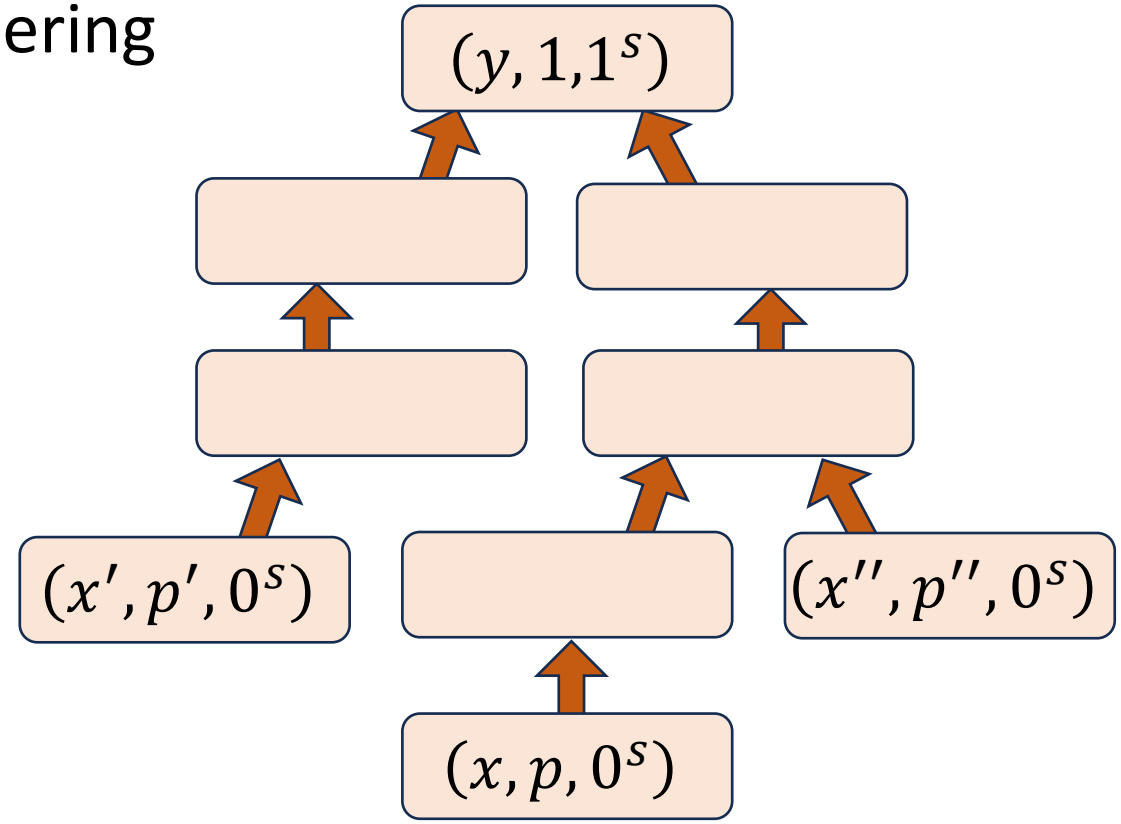
Theorem: There is a two-way numbering such that the rotation map $\mathrm{Rot}: V\times[D] \to V\times[D]$ and its inverse can both be computed in-place using $s' = s + O(\log D)$ bits of scratch space: $(x, p, 0^{s'}) \leftrightarrow (y, q, 1^{s'})$.

Two-way navigator: Proof sketch

Configurations from which $(y, 1, 1^s)$ is reachable form a tree $\mathcal{T}$

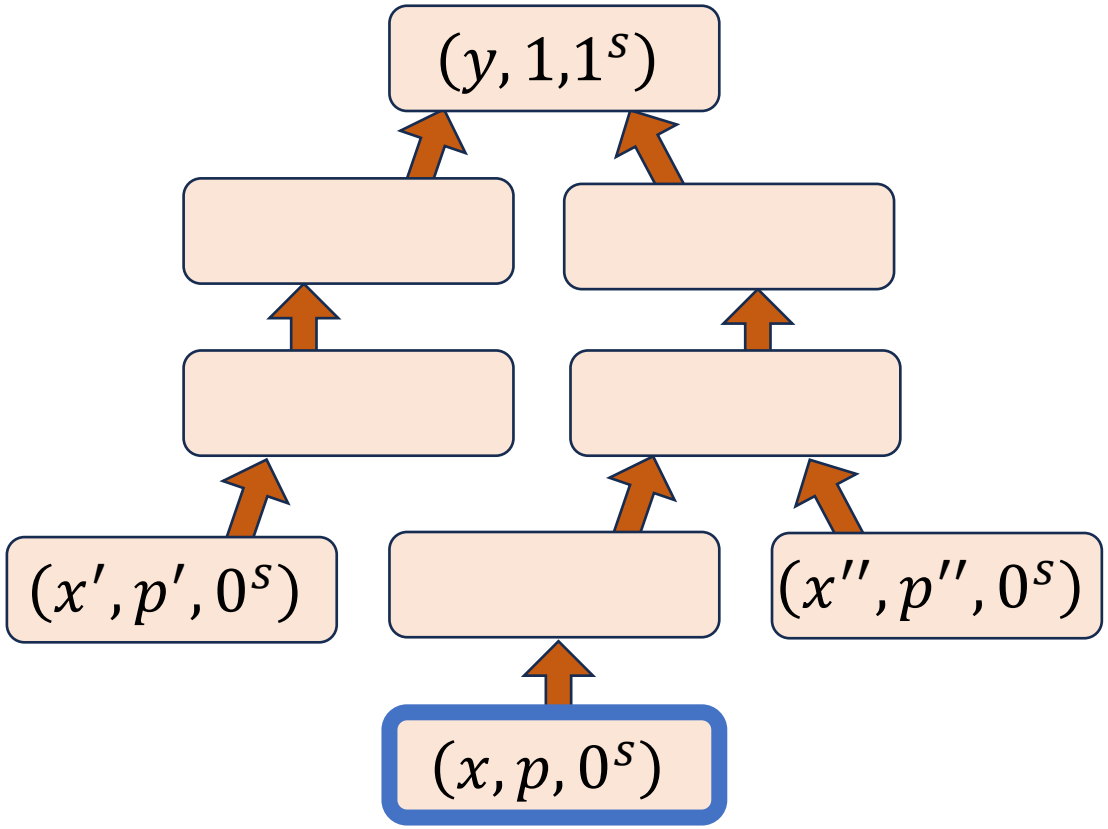
- Consider the computation graph of the Turing machine computing the one-way numbering





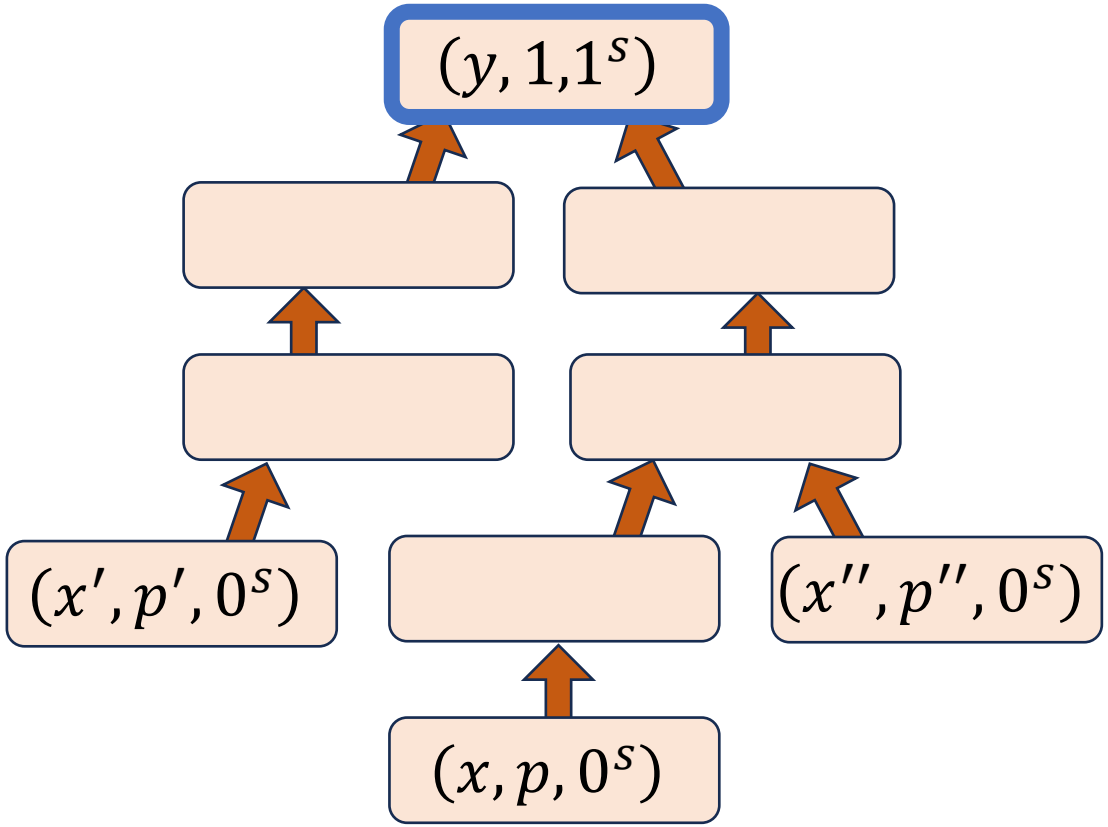
Two-way navigator: Proof sketch

- To compute $\mathrm{Rot}(x, p)$, do a depth-first search starting from $(x, p, 0^s)$
- $\mathrm{Rot}(x, p) = (y, q)$, where q is the number of $(_, _, 0^s)$ configurations visited before entering $(y, 1, 1^s)$ via final incoming edge



Two-way navigator: Proof sketch

- Use locality of Turing machine computation to find predecessor configurations!
- Space overhead:
  - $O(\log D)$ bits to count number of $(__, __, 0^s)$ configurations visited
  - $O(1)$ bits for ports in computation graph



Some difficulties we have glossed over

Difficulty 1: Ensuring sparsity

Ensuring that sparsified graphs are sparse

- **Definition (v1):** For $u, t \in [0,1]$ and distinct $X, Y \in \{T, U, V\}$, let

$$C^{X_u, Y_t}(x, y; \sigma) = C(x, y) \cdot 1[x \in X_u] \cdot 1[y \in Y_t]$$

$$E^{X_u, Y_t}(x, y; \sigma) = E(x, y) \cdot 1[x \in X_u] \cdot 1[y \in Y_t]$$

- So far, we've optimistically assumed C^{X_f, Y_f} and E^{X_f, Y_f} have degree $O(1)$

- Now it is time to face reality

Out-degree of E

- Recall: $E(x,y) = (C(x,y) - f/2)_+$

  [ignoring chop #2]

- $E(x,y) > 0 \Rightarrow C(x,y) > f/2$

- $\deg^+(E) \leq O(1/f)$

- A vertex in E^{X_f, Y_f} typically has $O(1)$ out-neighbors, but not always 😟

🌠 Wish:

- E has out-degree $O(1)$
- E has in-degree $O(1)$
- C has out-degree $O(1)$

“Row gatekeeper”



- Say that x is “good” if $|\{y \in Y_t : E(x,y) > 0\}| \leq 1000$
- Let $q_{x,y}^{X_u,Y_t} = \Pr[\, x \text{ is good} \mid x \in X_u, y \in Y_t \,] \geq 0.99$
- **Definition (v2):** $E^{X_u,Y_t}(x,y;\sigma) = E(x,y) \cdot 1[x \in X_u] \cdot 1[y \in Y_t] \cdot \frac{1[x \text{ is good}]}{q_{x,y}^{X_u,Y_t}}$
- This way, for every σ, we have $\deg^+(E^{X_u,Y_t}) \leq 1000$ ✅

Out-degree of C



🌠 Wish:

- E has out-degree $O(1)$
- E has in-degree $O(1)$
- C has out-degree $O(1)$

- Recall simplifying assumption: Every nonzero edge weight in C is $\Theta(f)$
- Therefore, $\deg^+(C) \leq O(1/f)$
- Use row gatekeeper to guarantee $\deg^+(C^{X_u,Y_t}) = O(1)$

In-degree of E

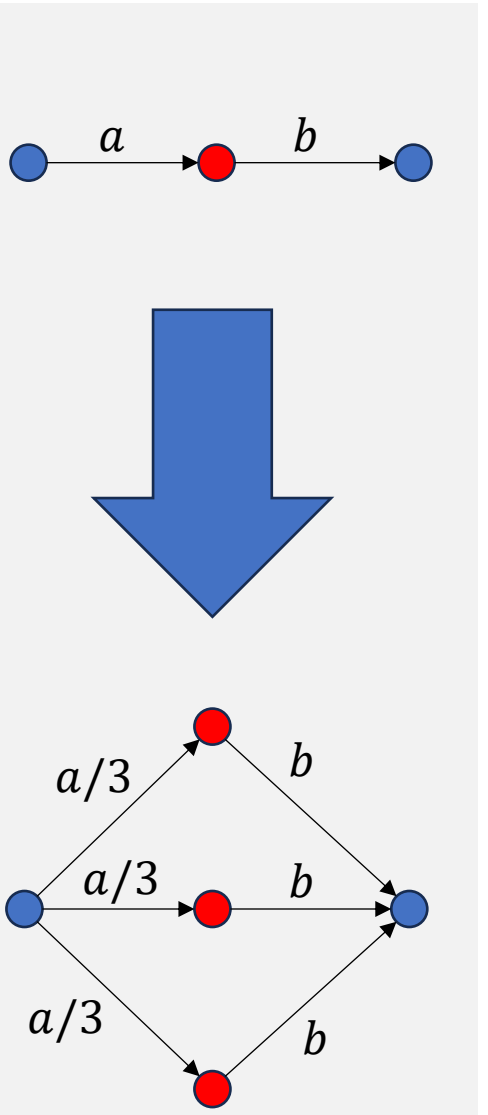
🌠 Wish:

- E has out-degree $O(1)$
- E has in-degree $O(1)$
- C has out-degree $O(1)$

- $\deg^+(E) \leq O(1/f)$
- So the average in-degree is $O(1/f)$…
- …but there are some “exceptional vertices” with high in-degree 😟

Solution: Splitting vertices

- Vertex y is "good" if $|\{x: C(x,y) > f/2\}| \leq 1000/f$
- **Definition (v2)**: $E(x,y) = (C(x,y) - f/2)_+ \cdot 1[y \text{ is good}]$
- **Definition (v2)**: Construct C' from $C - E + EC$ by splitting each bad vertex into a "cloud" of $O(1)$ vertices
- $\|C\|_{\max}$ still decreases from f down to $5f/6$, and # vertices only increases by a factor of 1.01



Shortcutting and splitting: Two distinct weapons



Photo by Wolfgang Sauber. License: CC-BY-SA 3.0

Linear algebra of vertex splitting

- **Goal:** Compute $C_0^0 + C_0^1 + \cdots + C_0^N = (I - C_0)^{-1}$
- **Strategy (v2):** $(I - C_0)^{-1} \approx A \cdot P_L \cdot (I + E_L) \cdots P_1 \cdot (I + E_1)$ where P_i splits bad vertices into clouds and A deletes all but one member of each cloud
- A and P_i are rectangular matrices

In-degree of E



🌠 Wish:

- E has out-degree $O(1)$
- E has in-degree $O(1)$
- C has out-degree $O(1)$

- Under new definition, $\deg^-(E) \leq O(1/f)$
- Each vertex in E^{X_u,Y_t} has expected in-degree $O(1)$
- Use a "column gatekeeper" to guarantee $\deg^-(E^{X_u,Y_t}) = O(1)$

Difficulty 2: Hard thresholding

Hard thresholding

- We have used hard thresholds in many places
- E.g., split vertices if # substantial incoming edges is more than 1000
- **Difficulty:** This rule is sensitive to tiny approximation errors
- **Solution:** Smooth out all the threshold rules
- E.g., always split vertices, but use a smooth rule to distribute incoming edge weight among copies

Conclusions

- Proof is complicated, but not beyond human comprehension
- Flavor is more “algorithm design” than “math”
- **Still open:** Design an explicit pseudorandom generator that stretches $O(\log n)$ truly random bits to n pseudorandom bits that can be used to simulate log-space computation

Conclusions

- **Thanks! Questions?**



Just for fun, here are some word frequencies in the OpenAI manuscript, expanding on observations by Jake Ruotolo

