CS 594 – Advanced Crypto (Spring 2026)

Alex Block

Lecture 22

April 13, 2026

Memory-hard Functions Continued

Memory-hard Functions: Recap

Memory-hard Functions: Recap

- A function f is *memory-hard* if the main computational costs are the amount of memory needed to compute f.

Memory-hard Functions: Recap

- A function f is *memory-hard* if the main computational costs are the amount of memory needed to compute f.

- Two flavors: data-*dependent* and data-*independent*.

Memory-hard Functions: Recap

- A function f is *memory-hard* if the main computational costs are the amount of memory needed to compute f.

- Two flavors: data-*dependent* and data-*independent*.
  - Data-dependent: memory-access pattern of $f(x)$ is different depending on input x.

Memory-hard Functions: Recap

- A function f is *memory-hard* if the main computational costs are the amount of memory needed to compute f.

- Two flavors: data-*dependent* and data-*independent*.
  - Data-dependent: memory-access pattern of $f(x)$ is different depending on input x.
  - Data-independent: memory-access pattern of $f(x)$ is fixed for any input x.

Memory-hard Functions: Recap

- A function f is *memory-hard* if the main computational costs are the amount of memory needed to compute f.

- Two flavors: data-*dependent* and data-*independent*.
  - Data-dependent: memory-access pattern of $f(x)$ is different depending on input x.
  - Data-independent: memory-access pattern of $f(x)$ is fixed for any input x.

- Both dMHFs and iMHFs are constructed using *directed acyclic graphs* G and a hash function H, modeled as a random oracle.

Measuring Memory Costs: Pebbling Arguments

Measuring Memory Costs: Pebbling Arguments

- Let $f_{G,H}$ be a function.

Measuring Memory Costs: Pebbling Arguments

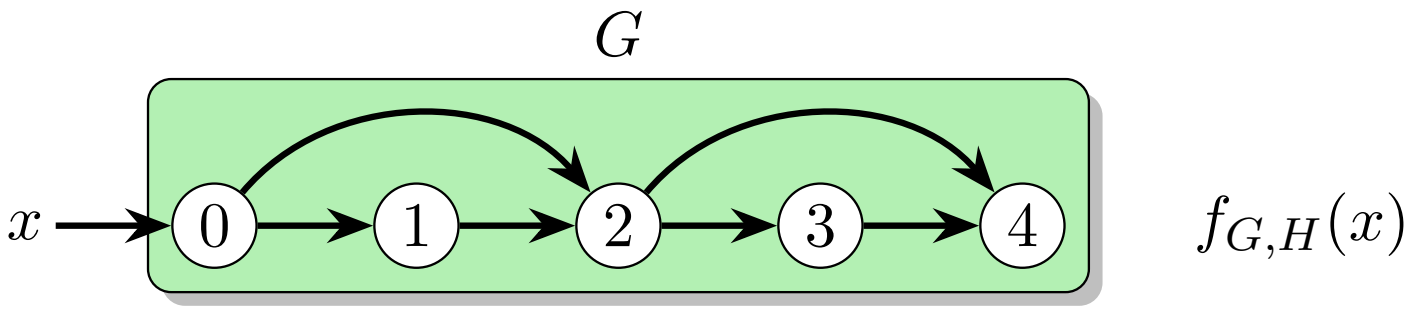
- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.

Measuring Memory Costs: Pebbling Arguments

- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.

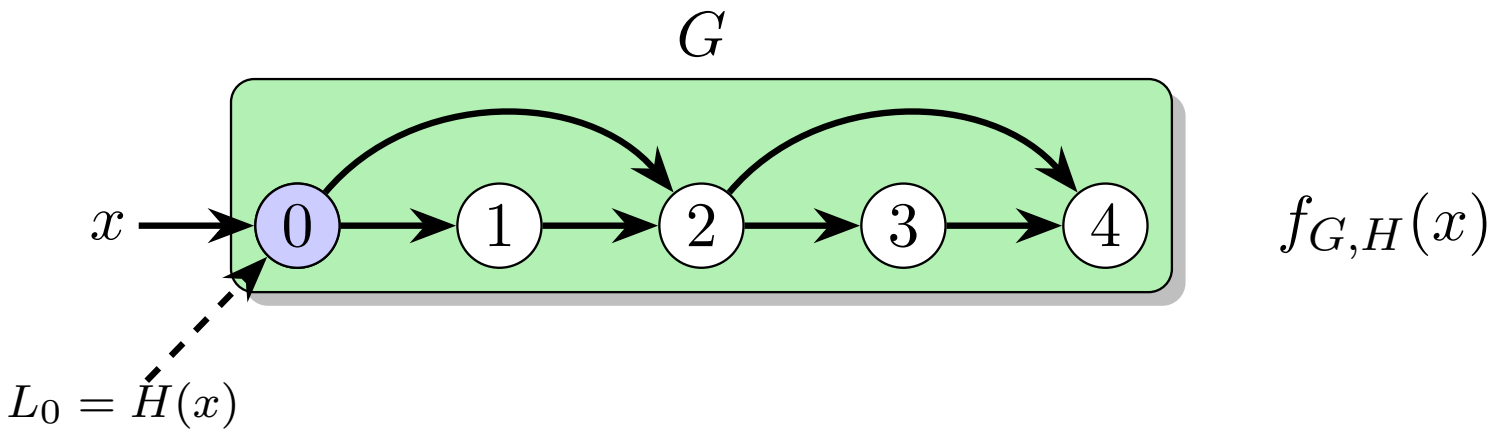
Measuring Memory Costs: Pebbling Arguments

- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



Measuring Memory Costs: Pebbling Arguments

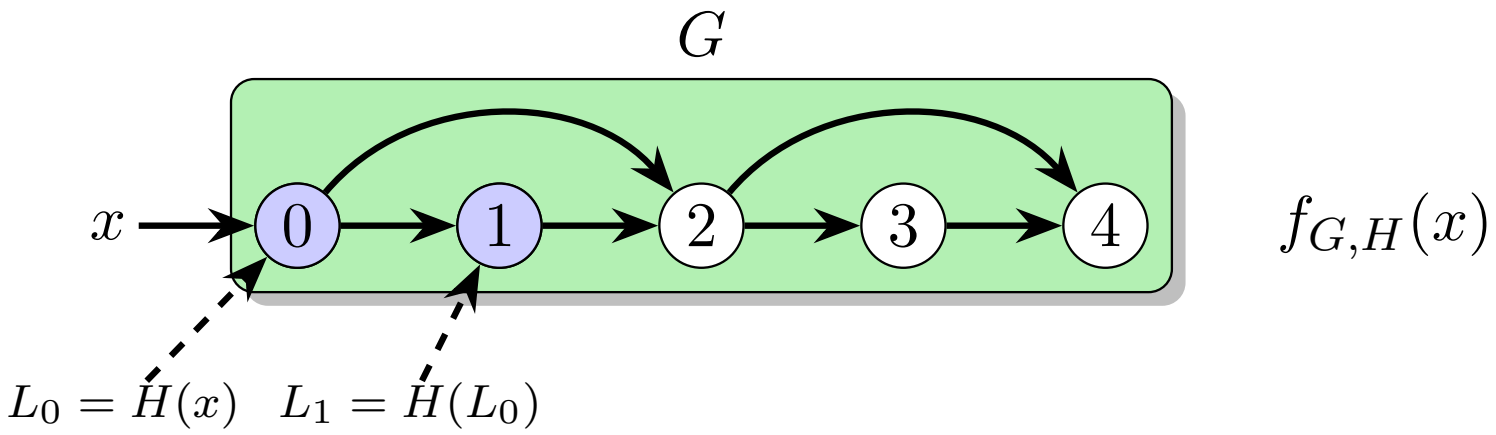
- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



- $P_0 = \{L_0\}$

Measuring Memory Costs: Pebbling Arguments

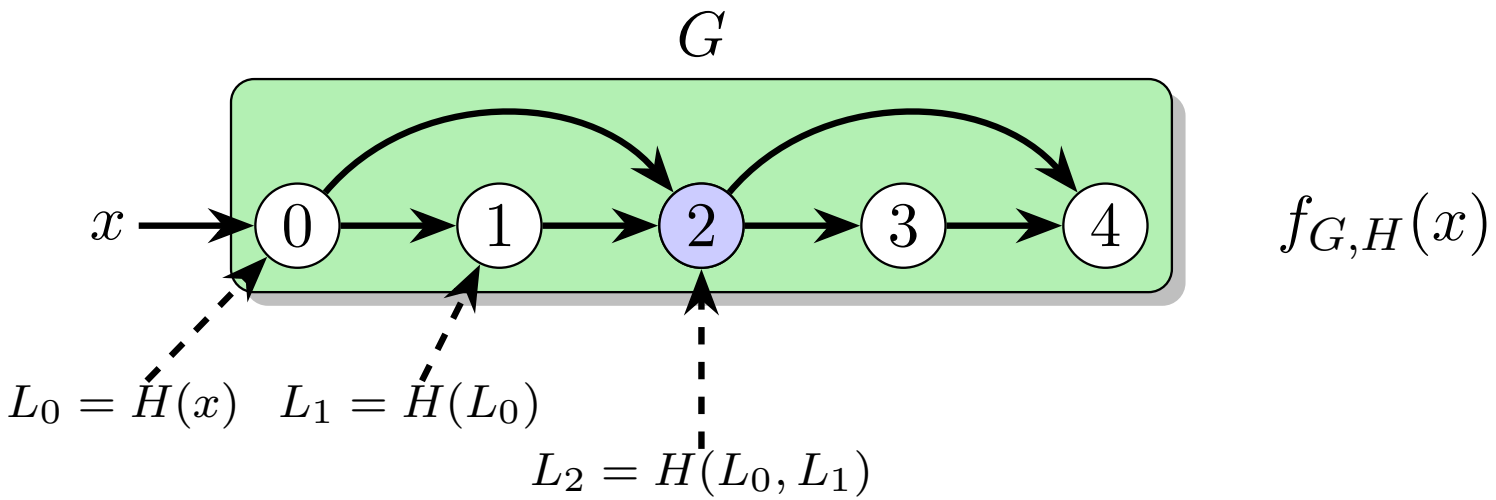
- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$

Measuring Memory Costs: Pebbling Arguments

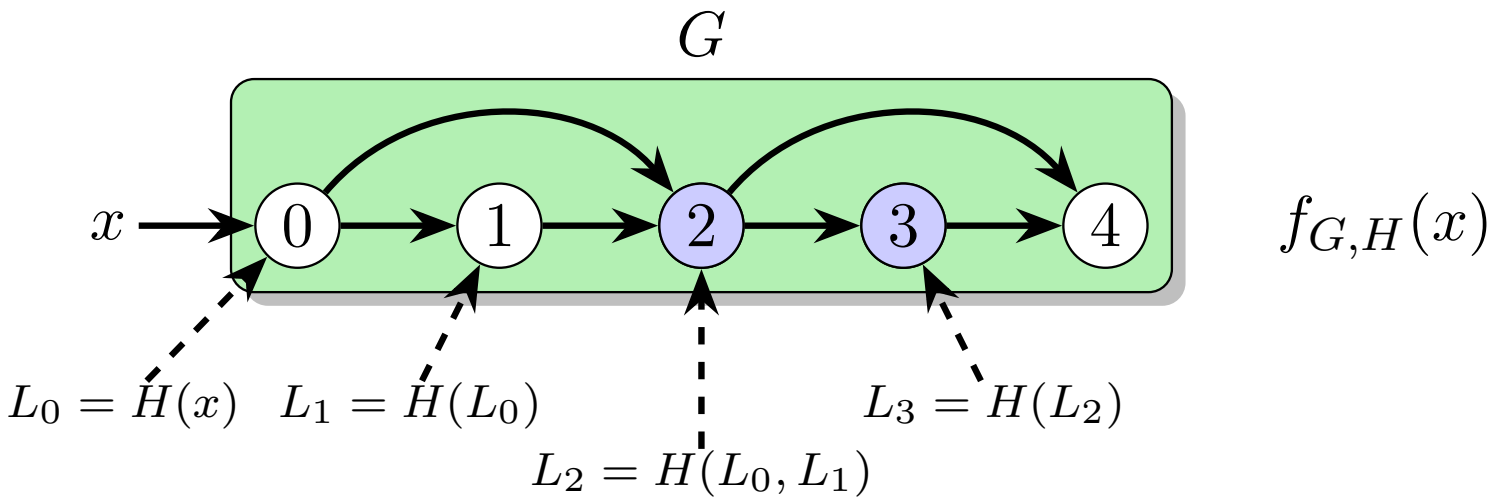
- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$

Measuring Memory Costs: Pebbling Arguments

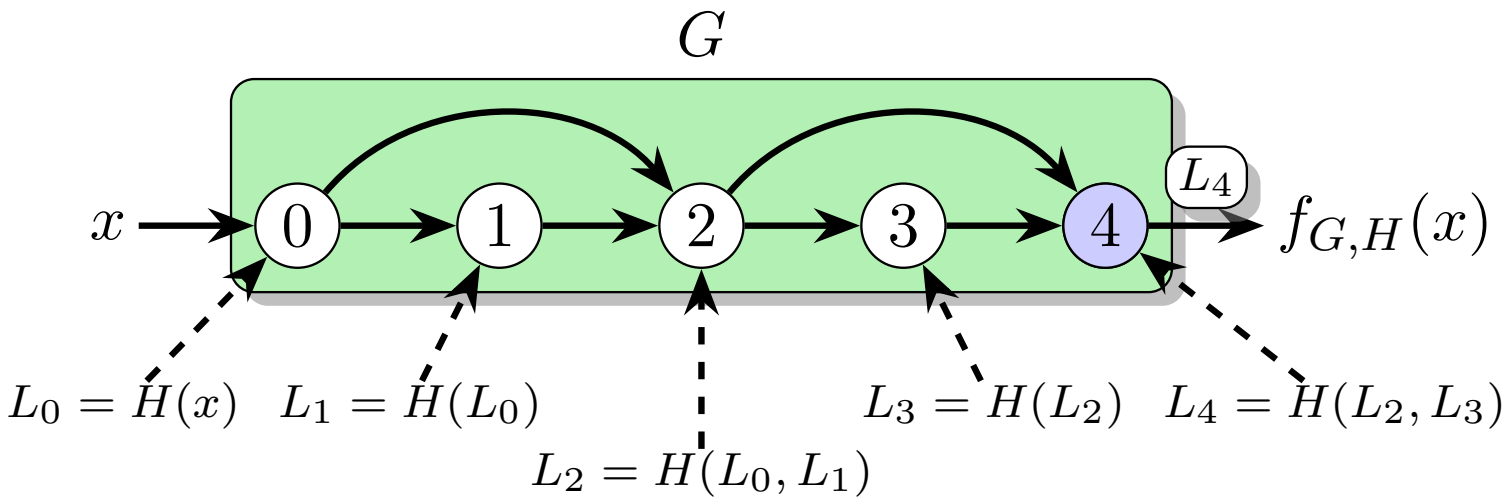
- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$
- $P_3 = \{L_2, L_3\}$

Measuring Memory Costs: Pebbling Arguments

- Let $f_{G,H}$ be a function.
- Goal: prove that $f_{G,H}$ is *memory-hard* for some notion of memory-hardness.
- Idea: memory-cost of $f_{G,H}$ is proportional to the *pebbling cost* of graph G with respect to H.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$
- $P_3 = \{L_2, L_3\}$
- $P_4 = \{L_4\}$

Measuring Memory Costs: ST Complexity

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.
- ST-Complexity has a rich theory and is studied in many works.

Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.
- ST-Complexity has a rich theory and is studied in many works.

ST-Complexity is not appropriate for password hashing!

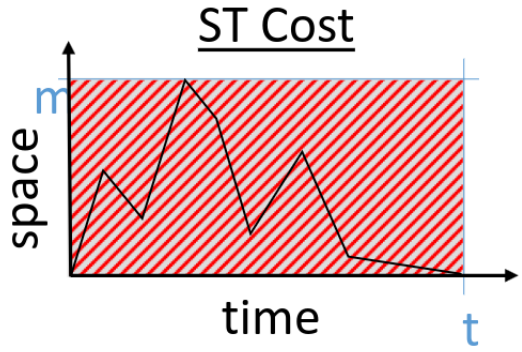
Measuring Memory Costs: ST Complexity

- First natural attempt: *space-time complexity*.
- For a DAG G, we define the *ST-Complexity* of G as

$$\mathsf{ST}(G) = \min_{\mathbf{P}} \left(T_{\mathbf{P}} \times \max_{i \leqslant T_{\mathbf{P}}} (|\mathbf{P}_i|) \right)$$

  - $\mathbf{P}$ is a pebbling strategy.
  - $T_{\mathbf{P}}$ is the time to compute $\mathbf{P}$.
  - $|\mathbf{P}_i|$ is the size of the i^{th} pebbling state.
- ST-Complexity has a rich theory and is studied in many works.

ST-Complexity is not appropriate for password hashing!



Amortization and Parallelism

Amortization and Parallelism

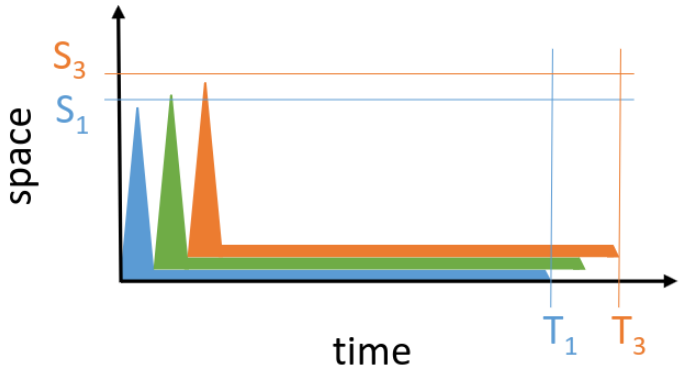
- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation*.

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation*.
  - Password hashing is a very parallel computation!

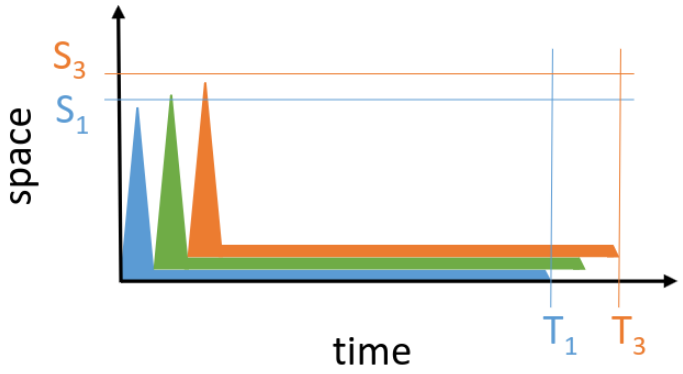
Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



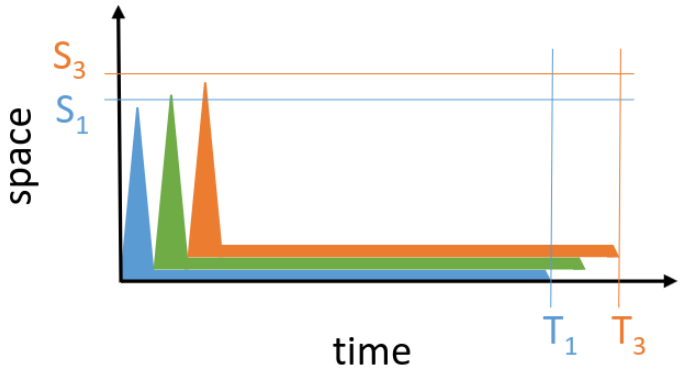
$$ST_1 = S_1 \times T_1 \approx S_3 \times T_3 = ST_3$$

cost of computing f once

cost of computing f three times

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



$$\mathsf{ST}_1 = S_1 \times T_1 \approx S_3 \times T_3 = \mathsf{ST}_3$$

cost of computing f once

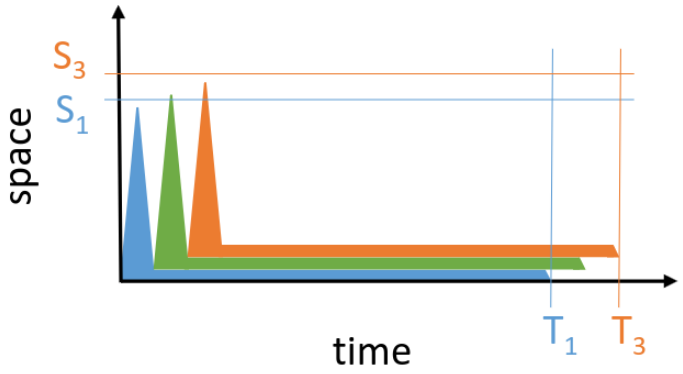
cost of computing f three times

Theorem 1

There exists a function f_n (making n calls to a random oracle) such that the ST-Complexity of $\sqrt{n}$ parallel calls to f_n is $O(\mathsf{ST}(f))$.

Amortization and Parallelism

- **Issue**: ST-Complexity can scale poorly in the number of function evaluations *for a parallel computation.*
  - Password hashing is a very parallel computation!



$$\mathsf{ST}_1 = S_1 \times T_1 \approx S_3 \times T_3 = \mathsf{ST}_3$$

cost of computing f once

cost of computing f three times

Theorem 1

There exists a function f_n (making n calls to a random oracle) such that the ST-Complexity of $\sqrt{n}$ parallel calls to f_n is $O(\mathsf{ST}(f))$.

For measuring memory-hardness, we would like an *amortized* memory cost (amortized area-time complexity)

Cumulative (Memory) Complexity

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.
- Guessing two passwords now doubles an attacker's cost!

Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.
- Guessing two passwords now doubles an attacker's cost!

$$\mathsf{CC}(G, G) = 2 \cdot \mathsf{CC}(G)$$

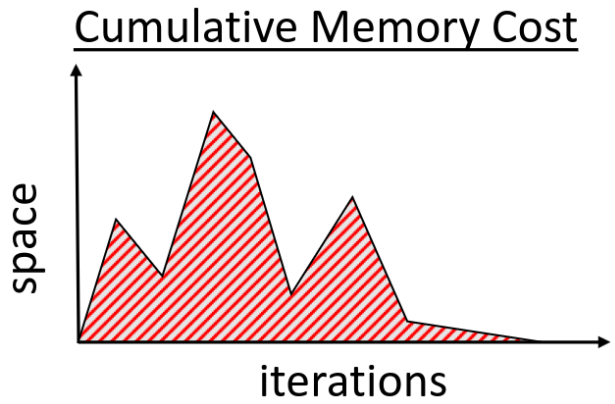
Cumulative (Memory) Complexity

- The notion of *cumulative (memory) complexity* (CC) was introduced to approximate the amortized area-time complexity of pebbling a DAG G.

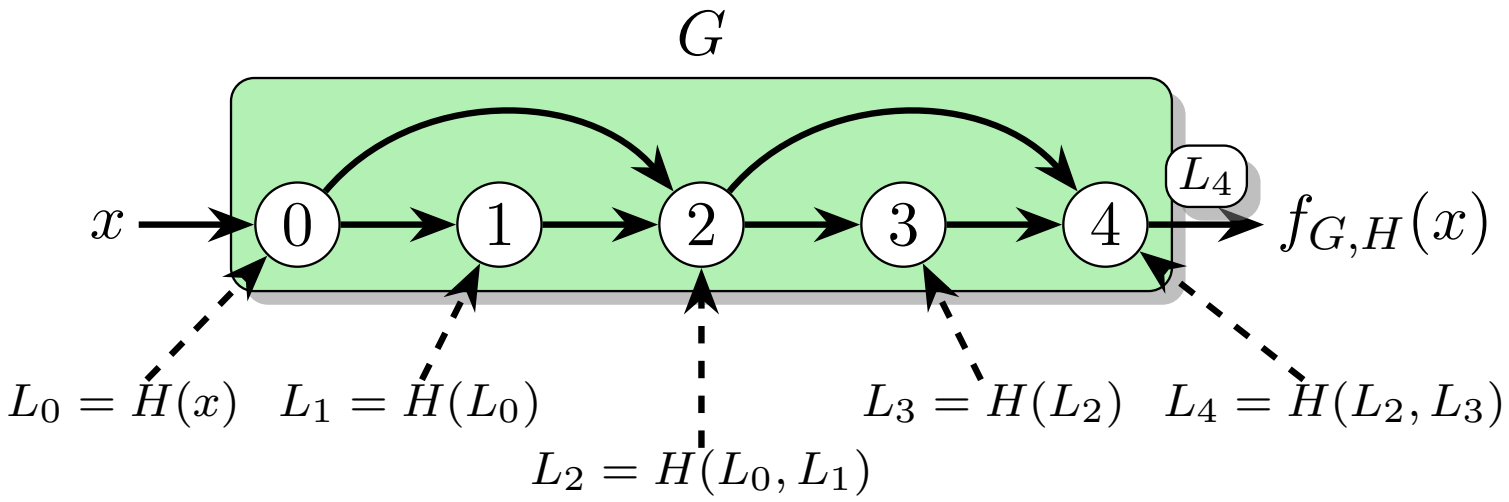
$$\mathsf{CC}(G) = \min_{\mathbf{P}} \sum_{i=1}^{T_{\mathbf{P}}} |\mathbf{P}_i|$$

- $|\mathbf{P}_i|$ is the memory used at step i of the computation.
- Guessing two passwords now doubles an attacker's cost!

$$\mathsf{CC}(G, G) = 2 \cdot \mathsf{CC}(G)$$

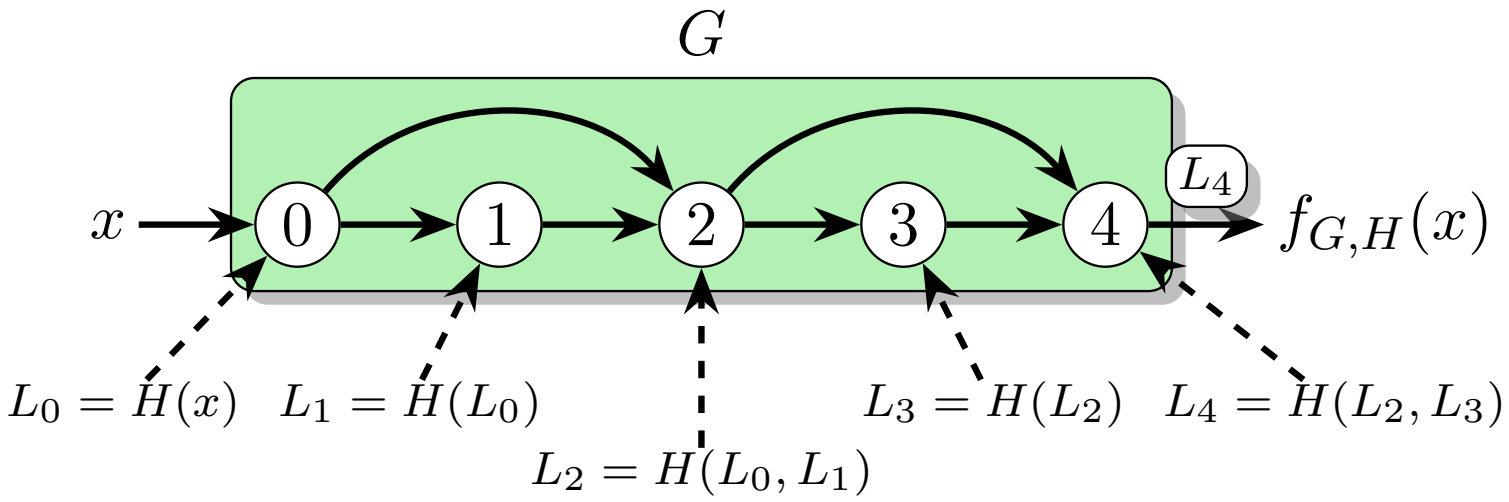


Pebbling Example: ST vs CC



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$
- $P_3 = \{L_2, L_3\}$
- $P_4 = \{L_4\}$

Pebbling Example: ST vs CC

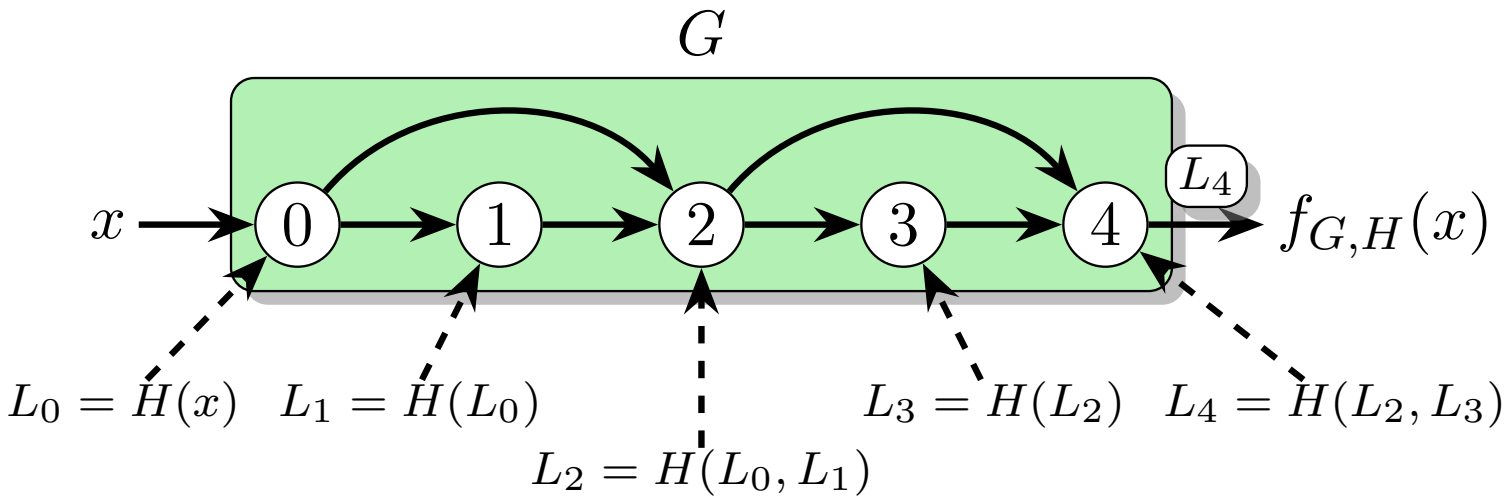


- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$
- $P_3 = \{L_2, L_3\}$
- $P_4 = \{L_4\}$

ST Complexity

$$\mathsf{ST}(G) = 5 \cdot 2 = 10$$

Pebbling Example: ST vs CC



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_2\}$
- $P_3 = \{L_2, L_3\}$
- $P_4 = \{L_4\}$

ST Complexity

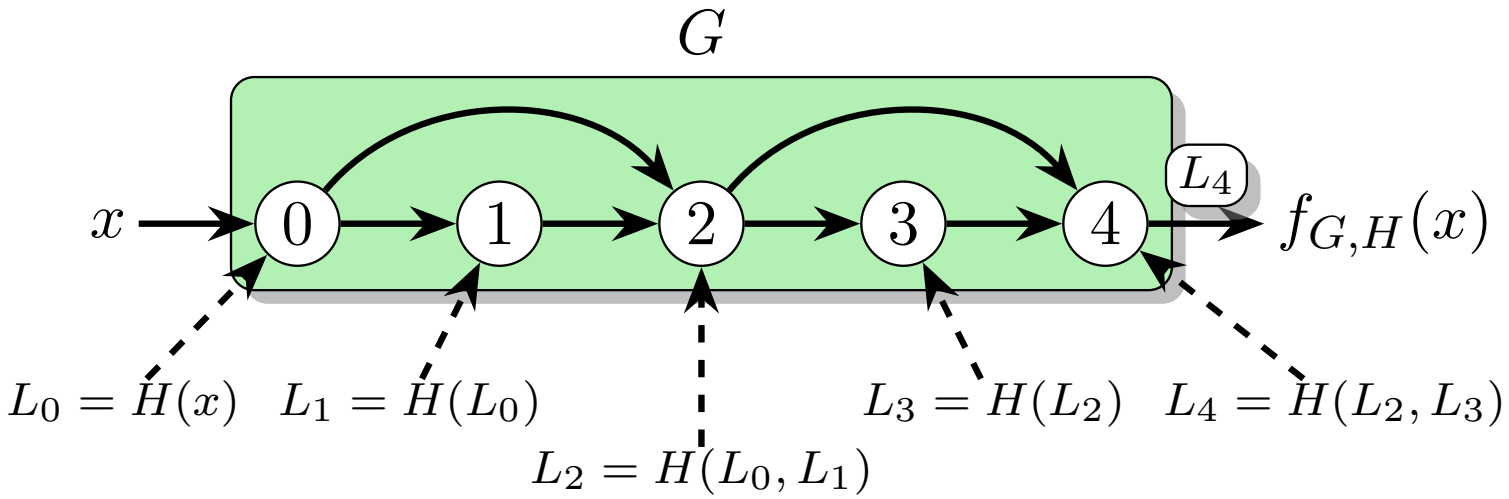
$$\mathsf{ST}(G) = 5 \cdot 2 = 10$$

CC

$$\mathsf{CC}(G) = 1 + 2 + 1 + 2 + 1 = 7$$

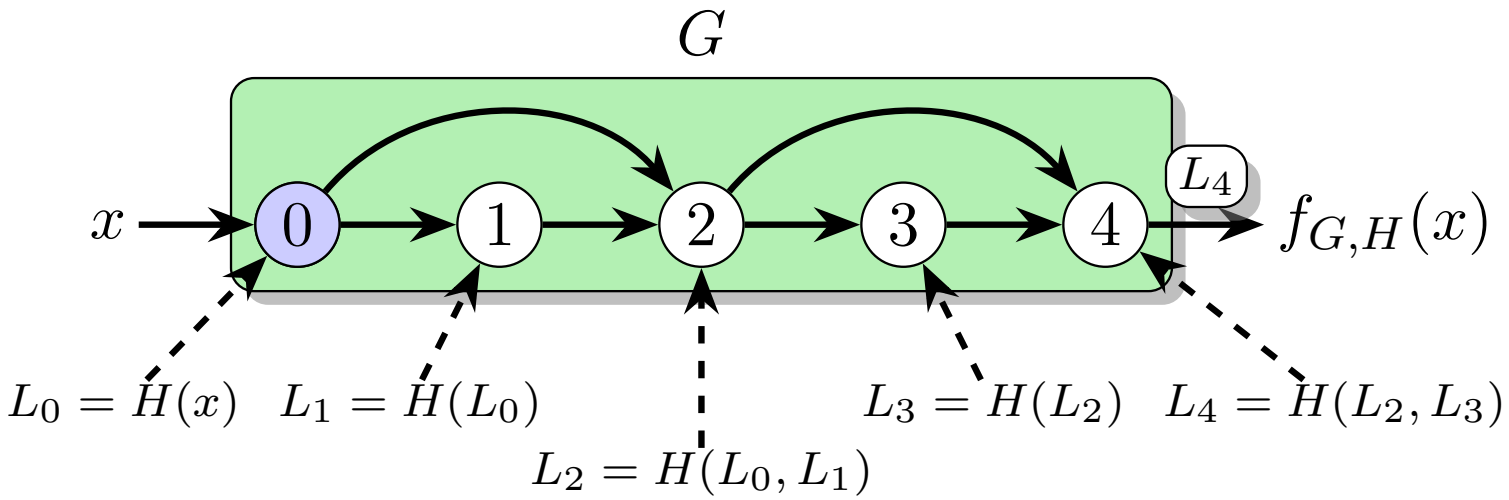
Naïve Pebbling Strategy Example

- There always exists the naïve pebbling strategy for any DAG.



Naïve Pebbling Strategy Example

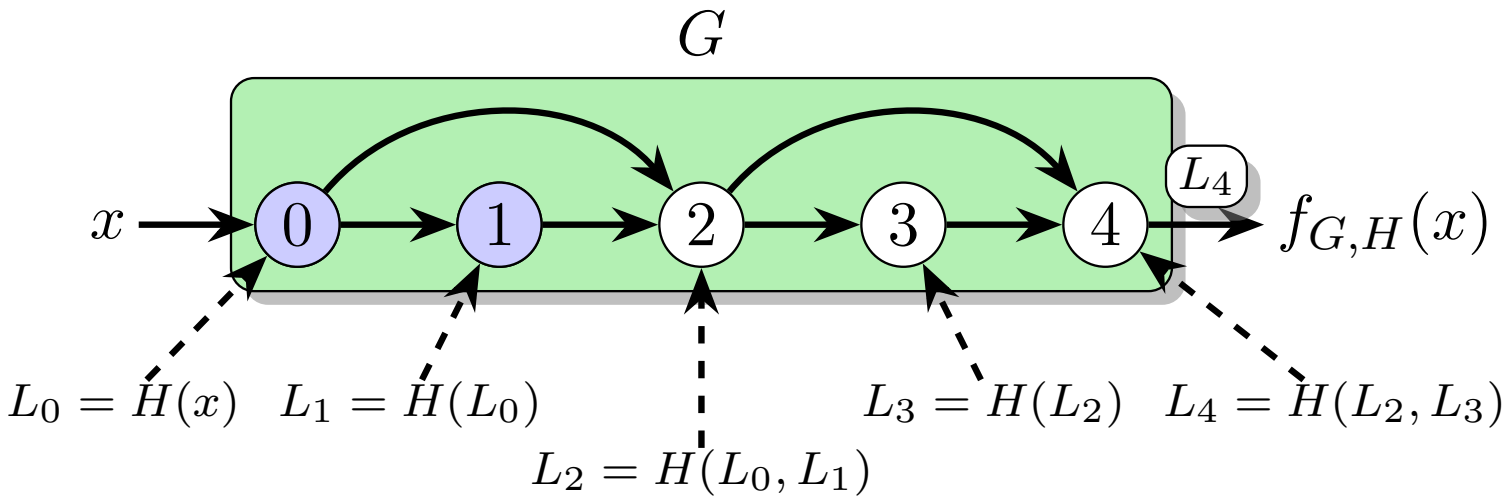
- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$

Naïve Pebbling Strategy Example

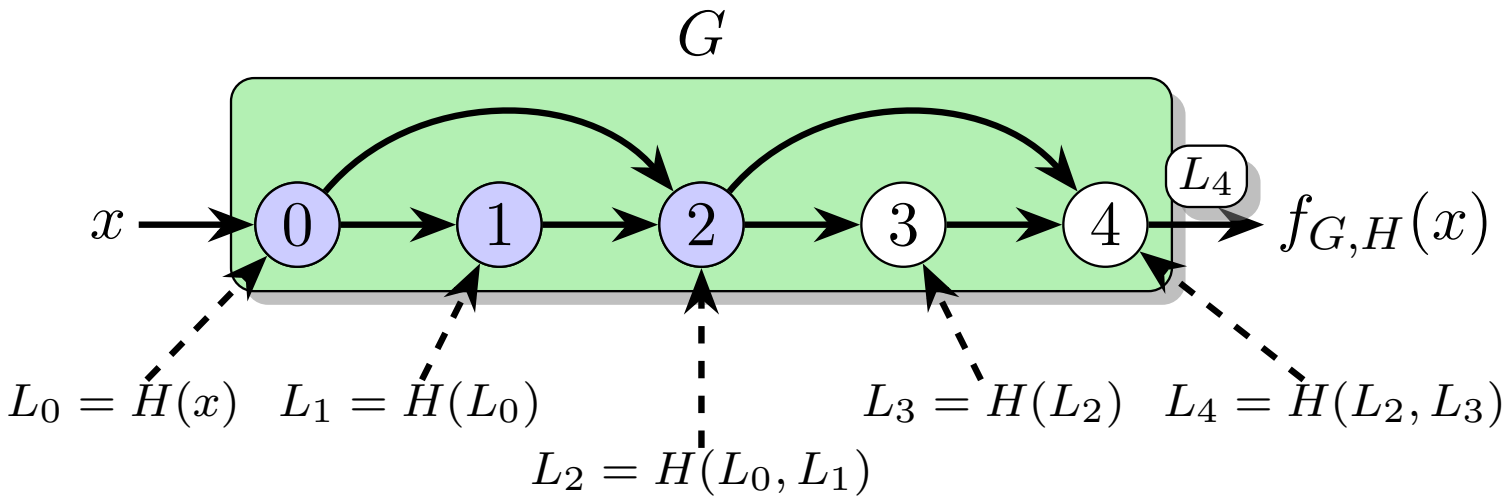
- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$

Naïve Pebbling Strategy Example

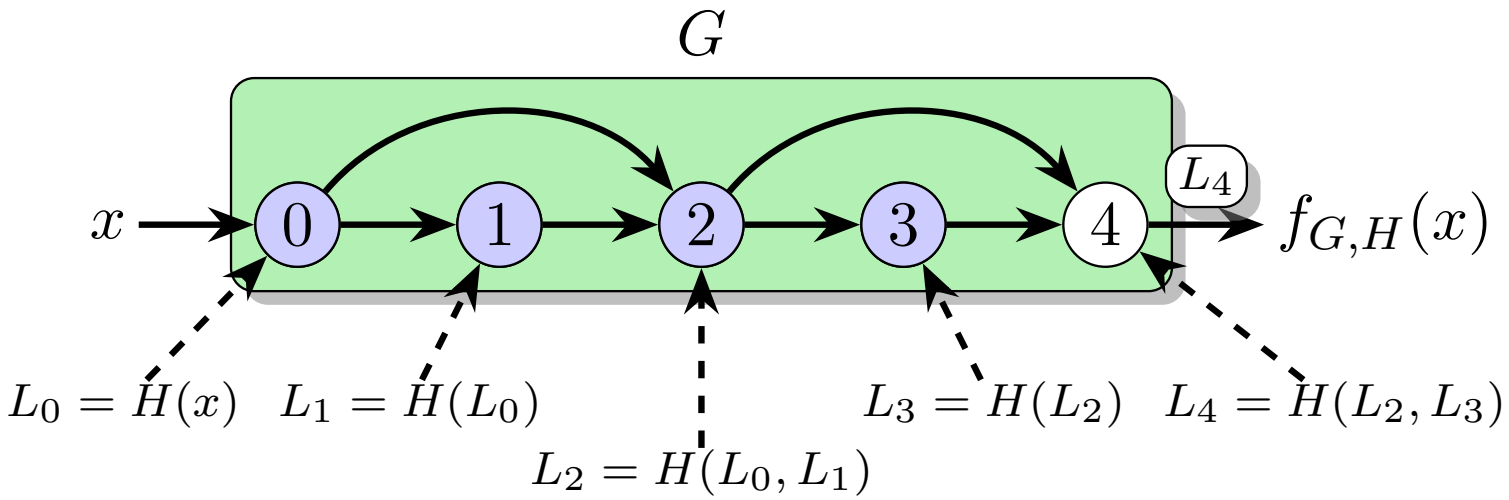
- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$

Naïve Pebbling Strategy Example

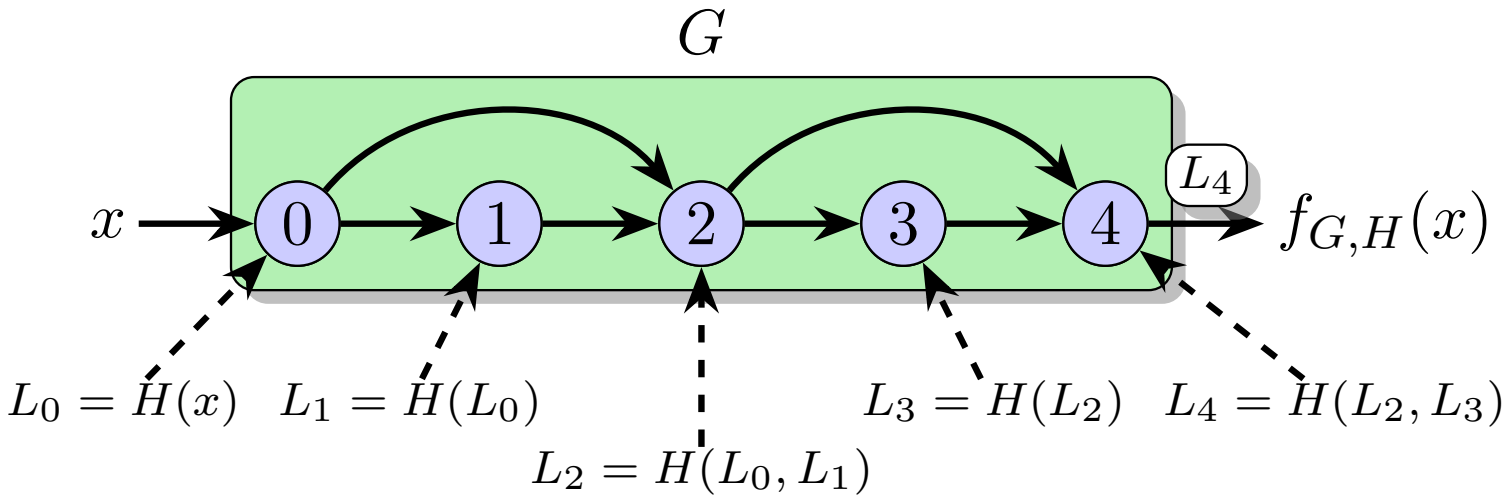
- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$

Naïve Pebbling Strategy Example

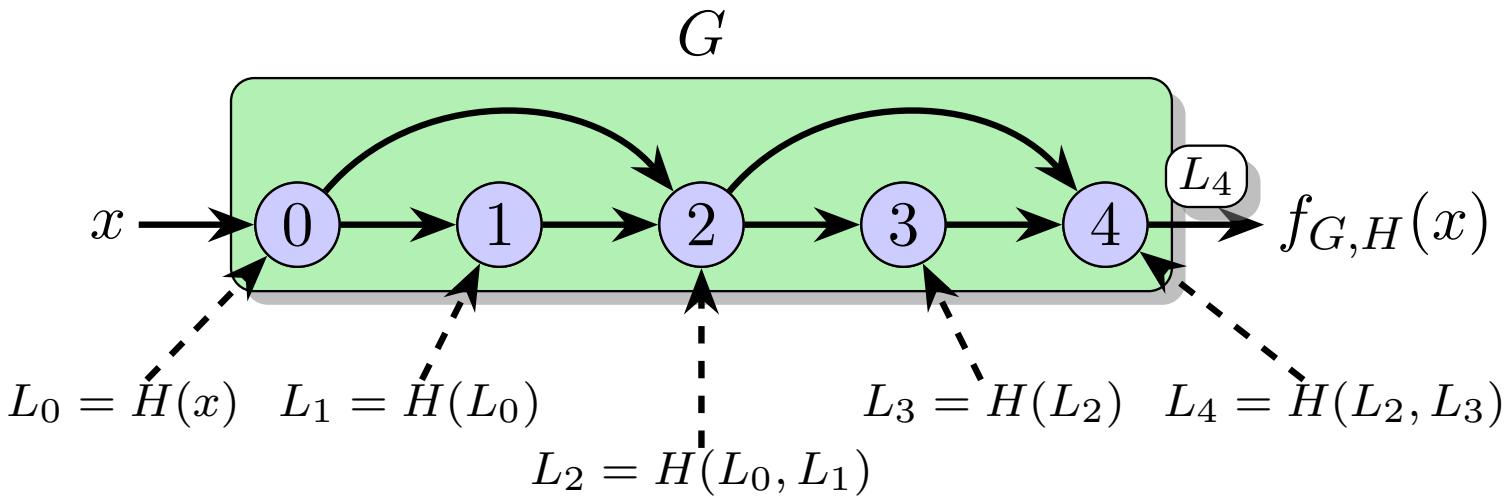
- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{\cancel{L_0, L_1, L_2, L_3}, L_4\}$

Naïve Pebbling Strategy Example

- There always exists the naïve pebbling strategy for any DAG.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_0, L_1, L_2, L_3, L_4\}$

CC

$$\mathsf{CC}(G, \mathbf{P}) = 1 + 2 + 3 + 4 + 5 = 15$$

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:
  - Legal for any DAG G.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:
  - Legal for any DAG G.
  - Pebbling time is n.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:
  - Legal for any DAG G.
  - Pebbling time is n.
  - Sequential algorithm: place a single new pebble on the graph during each round of pebbling.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:
  - Legal for any DAG G.
  - Pebbling time is n.
  - Sequential algorithm: place a single new pebble on the graph during each round of pebbling.

Theorem 2

Any DAG G on n nodes has $\mathsf{CC}(G) \leqslant \sum_{i \leqslant n} i = \frac{n(n+1)}{2}$.

Naïve Pebbling Strategy

- The naïve pebbling strategy can be described as follows:
  - Pebble the DAG G in topological order.
  - Never discard any pebbles.

- Properties of naïve pebbling strategy:
  - Legal for any DAG G.
  - Pebbling time is n.
  - Sequential algorithm: place a single new pebble on the graph during each round of pebbling.

Theorem 2

Any DAG G on n nodes has $\mathsf{CC}(G) \leqslant \sum_{i \leqslant n} i = \frac{n(n+1)}{2}$.

Proof.

The naïve pebbling strategy is legal for all DAGs! □

Graphs with Large CC

- This theorem tells us that $O(n^2)$ is the upper bound on CC for all DAGs.

Graphs with Large CC

- This theorem tells us that $O(n^2)$ is the upper bound on CC for all DAGs.

Question

Does there exist any DAG G with $\mathsf{CC}(G) = \Omega(n^2)$?

Graphs with Large CC

- This theorem tells us that $O(n^2)$ is the upper bound on CC for all DAGs.

Question

Does there exist any DAG G with $\mathsf{CC}(G) = \Omega(n^2)$?

- Such a graph would be *optimally memory-hard* (with respect to CC).

Graphs with Large CC

- This theorem tells us that $O(n^2)$ is the upper bound on CC for all DAGs.

Question

Does there exist any DAG G with $\mathsf{CC}(G) = \Omega(n^2)$?

- Such a graph would be *optimally memory-hard* (with respect to CC).

Yes!

The complete DAG on n vertices has large CC.

Graphs with Large CC

- This theorem tells us that $O(n^2)$ is the upper bound on CC for all DAGs.

Question

Does there exist any DAG G with $\mathsf{CC}(G) = \Omega(n^2)$?

- Such a graph would be *optimally memory-hard* (with respect to CC).

Yes!

The complete DAG on n vertices has large CC.

But!

This graph has large in-degree $O(n)$!

Graphs with Large CC

Graphs with Large CC

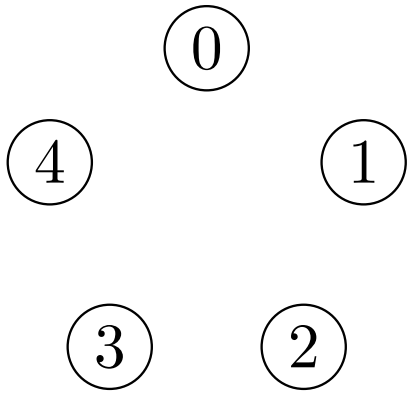
Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2}$ $= \Omega(n^2)$.

Graphs with Large CC

Claim 1

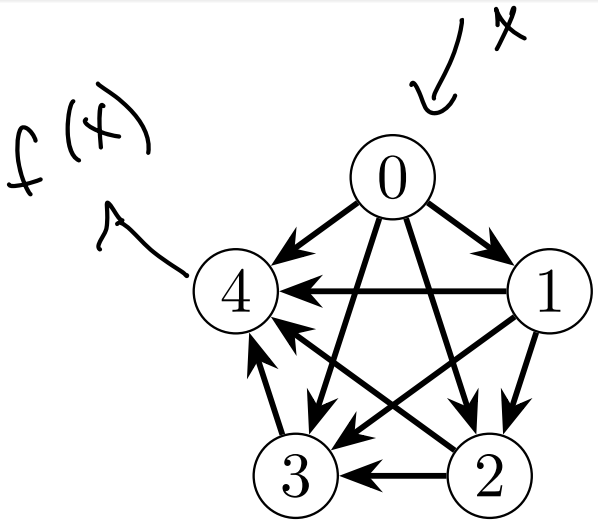
The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2}$ $= \Omega(n^2)$.



Graphs with Large CC

Claim 1

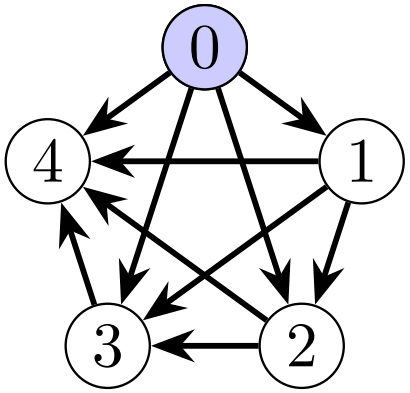
The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2}$ $= \Omega(n^2)$.



Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.

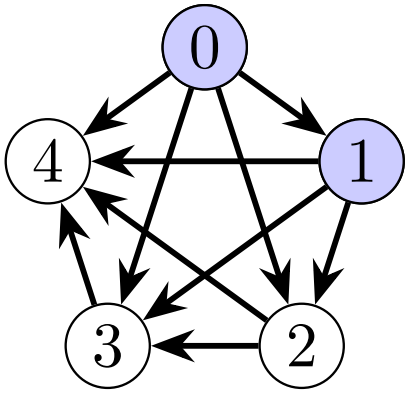


- $P_0 = \{L_0\}$

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.

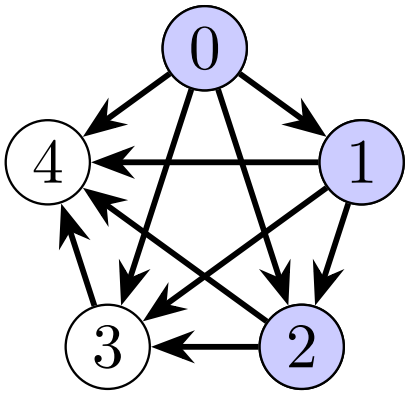


- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.

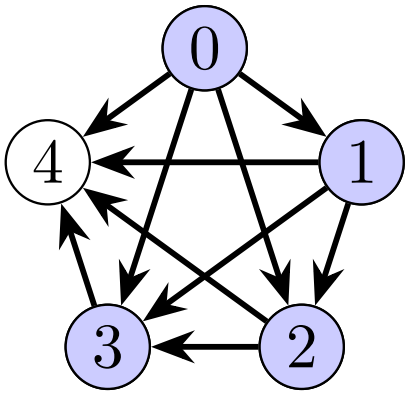


- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.

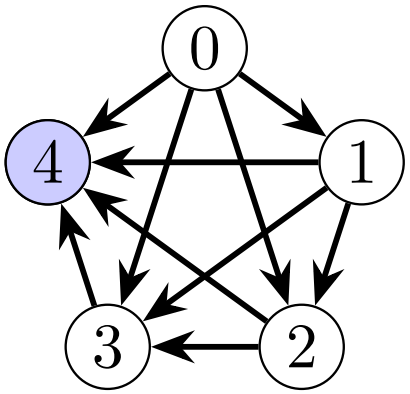


- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.

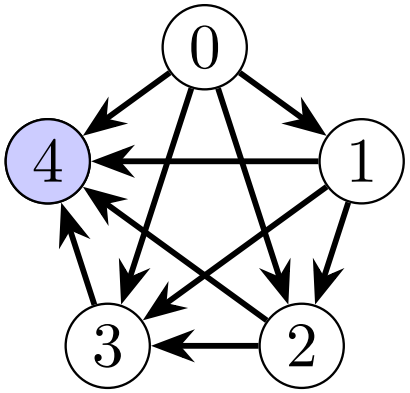


- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_4\}$

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_4\}$

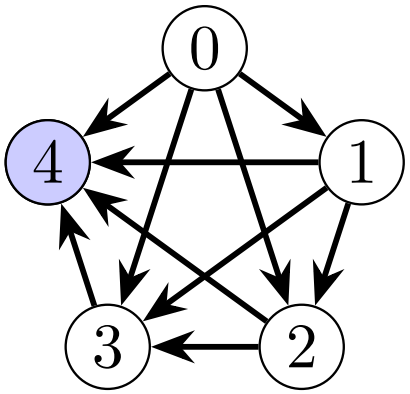
Proof.

□

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_4\}$

Proof.

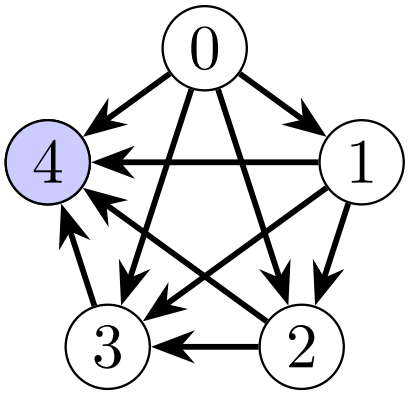
Label the graph from 0 to $n-1$ in topological order.

□

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_4\}$

Proof.

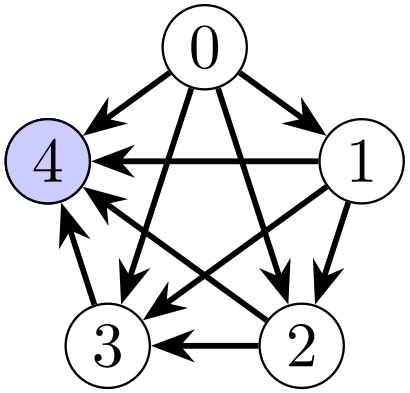
Label the graph from 0 to $n-1$ in topological order. Then, node i has parent nodes $\{0, \ldots, i-1\}$.

□

Graphs with Large CC

Claim 1

The complete DAG on n vertices has $\mathsf{CC}(G) \geqslant \sum_{i \leqslant n-1} i = \frac{n(n-1)}{2} = \Omega(n^2)$.



- $P_0 = \{L_0\}$
- $P_1 = \{L_0, L_1\}$
- $P_2 = \{L_0, L_1, L_2\}$
- $P_3 = \{L_0, L_1, L_2, L_3\}$
- $P_4 = \{L_4\}$

Proof.

Label the graph from 0 to $n-1$ in topological order. Then, node i has parent nodes $\{0, \ldots, i-1\}$. So node i needs all of its parents to be pebbled before it can be pebbled. □

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h : \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$. In practice, this is $h(h(L_u, L_w), L_x)$, or 2 calls to h.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$. In practice, this is $h(h(L_u, L_w), L_x)$, or 2 calls to h.
  - If v has k parents, then we must make $k - 1$ calls to h.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$. In practice, this is $h(h(L_u, L_w), L_x)$, or 2 calls to h.
  - If v has k parents, then we must make $k - 1$ calls to h.
- Runtime of $f_{G,h}$ is proportional to $n \cdot \delta(G)$.

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$. In practice, this is $h(h(L_u, L_w), L_x)$, or 2 calls to h.
  - If v has k parents, then we must make $k-1$ calls to h.
- Runtime of $f_{G,h}$ is proportional to $n \cdot \delta(G)$.
  - $\delta(G) = \log(n)$ already gives us super-linear runtime!

Why does In-degree Matter?

Question

Can we find a DAG G with $\mathsf{CC}(G) = \Omega(n^2)$ and small in-degree? I.e., $\delta(G) = O(1)$?

- Why do we care?
- In practice, the random oracle H is instantiated as a fixed hash function $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- To get a label of node v in the graph, we must do the following.
  - If v has 2 parents u, w, then $L_v = H(L_u, L_w)$, one oracle call and one call to h.
  - If v has 3 parents u, w, x, then $L_v = H(L_u, L_w, L_x)$. In practice, this is $h(h(L_u, L_w), L_x)$, or 2 calls to h.
  - If v has k parents, then we must make $k-1$ calls to h.
- Runtime of $f_{G,h}$ is proportional to $n \cdot \delta(G)$.
  - $\delta(G) = \log(n)$ already gives us super-linear runtime!
  - Bad for users!

Desiderata for iMHF DAGs

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

- Goal: find a fixed DAG G on n nodes such that:

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

- Goal: find a fixed DAG G on n nodes such that:
  1. $\delta(G) = 2$ (constant in-degree)

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

- Goal: find a fixed DAG G on n nodes such that:
  1. $\delta(G) = 2$ (constant in-degree)
     - Gives us a running time of $n(\delta - 1) = n$ for computing $f_{G,H}$.

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

- Goal: find a fixed DAG G on n nodes such that:
  1. $\delta(G) = 2$ (constant in-degree)
     - Gives us a running time of $n(\delta - 1) = n$ for computing $f_{G,H}$.
  2. $\mathsf{CC}(G) \geqslant n^2/\tau$ for some small value τ. constant

Desiderata for iMHF DAGs

- Reminder: key difference between dMHF and iMHF DAGs:
  - iMHF DAG: fixed ahead of time.
  - dMHF DAG: nodes are fixed, but parents of a node are randomly sampled as part of the computation process (sampling depends on the input).

- Goal: find a fixed DAG G on n nodes such that:
  1. $\delta(G) = 2$ (constant in-degree)
     - Gives us a running time of $n(\delta - 1) = n$ for computing $f_{G,H}$.
  2. $\mathsf{CC}(G) \geqslant n^2/\tau$ for some small value τ.

Maximize the costs for fixed running time n (users are impatient!)

iMHFs cannot have good CC

iMHFs cannot have good CC

- Unfortunately, fixed DAGs cannot have optimal CC.

with const. indegree

iMHFs cannot have good CC

- Unfortunately, fixed DAGs cannot have optimal CC.

Theorem 3 (Alwen-Blocki 2016)

For all DAGs G on n vertices such that $\delta(G) = O(1)$, it holds that
$\mathsf{CC}(G) = O\left(\frac{n^2 \operatorname{loglog}(n)}{\log(n)}\right) = \omega(n^2)$

iMHFs cannot have good CC

- Unfortunately, fixed DAGs cannot have optimal CC.

Theorem 3 (Alwen-Blocki 2016)

For all DAGs G on n vertices such that $\delta(G) = O(1)$, it holds that $\mathsf{CC}(G) = O\left(\frac{n^2 \operatorname{loglog}(n)}{\log(n)}\right)$

- However, there are some graph constructions that nearly match this upper bound.

iMHFs cannot have good CC

- Unfortunately, fixed DAGs cannot have optimal CC.

Theorem 3 (Alwen-Blocki 2016)

For all DAGs G on n vertices such that $\delta(G) = O(1)$, it holds that $\mathsf{CC}(G) = O\left(\frac{n^2 \operatorname{loglog}(n)}{\log(n)}\right)$

- However, there are some graph constructions that nearly match this upper bound.

Theorem 4

There exists a DAG G on n vertices with $\delta(G) = O(1)$ such that $\mathsf{CC}(G) \geqslant \Omega\left(\frac{n^2}{\log(n)}\right)$.

CC, CMC, AND THE PROM

CC and Memory Costs

CC and Memory Costs

- So far, we have only been discussing memory costs intuitively.

CC and Memory Costs

- So far, we have only been discussing memory costs intuitively.

- Intuition: $\mathsf{CC}(G)$ should correspond to the amount of memory used by an *algorithm* A computing $f_{G,H}$.

CC and Memory Costs

- So far, we have only been discussing memory costs intuitively.
- Intuition: $\mathsf{CC}(G)$ should correspond to the amount of memory used by an *algorithm* A computing $f_{G,H}$.
- Goal: formalize this intuition.

CC and Memory Costs

- So far, we have only been discussing memory costs intuitively.
- Intuition: $\mathsf{CC}(G)$ should correspond to the amount of memory used by an *algorithm* A computing $f_{G,H}$.
- Goal: formalize this intuition.
- For an algorithm A computing some function f, we will define the notion of *cumulative memory complexity* (cmc).

CC and Memory Costs

- So far, we have only been discussing memory costs intuitively.
- Intuition: $\mathsf{CC}(G)$ should correspond to the amount of memory used by an *algorithm* A computing $f_{G,H}$.
- Goal: formalize this intuition.
- For an algorithm A computing some function f, we will define the notion of *cumulative memory complexity* (cmc).
  - Intuition: cmc of A computing f on input x is the summation of the amount of memory used by A during every step of the computation.

Cumulative Memory Complexity

Cumulative Memory Complexity

- Since our model for MHFs utilizes a random oracle H, we need to define or memory model and costs with respect to a random oracle.

Cumulative Memory Complexity

- Since our model for MHFs utilizes a random oracle H, we need to define or memory model and costs with respect to a random oracle.
- Let A_f^H be a *randomized* algorithm computing f for any input x that is allowed to make *parallel* calls to random oracle H.

Cumulative Memory Complexity

- Since our model for MHFs utilizes a random oracle H, we need to define or memory model and costs with respect to a random oracle.
- Let A_f^H be a *randomized* algorithm computing f for any input x that is allowed to make *parallel* calls to random oracle H.
- Let $\text{Trace}(A_f^H, x) = (\sigma_0, \sigma_1, \ldots, \sigma_T)$ be the *execution trace* of $A_f^H(x)$ terminating in T steps.

Cumulative Memory Complexity

- Since our model for MHFs utilizes a random oracle H, we need to define or memory model and costs with respect to a random oracle.
- Let A_f^H be a *randomized* algorithm computing f for any input x that is allowed to make *parallel* calls to random oracle H.
- Let $\text{Trace}(A_f^H, x) = (\sigma_0, \sigma_1, \ldots, \sigma_T)$ be the *execution trace* of $A_f^H(x)$ terminating in T steps.

Definition 1 (cmc)

The *cumulative memory complexity (cmc)* of A is defined as

$$\mathsf{cmc}(A_f) = \mathop{\mathbb{E}}_{H}\left[\min_{x,r} \sum_{\sigma \in \text{Trace}(A_f^H(r),x)} |\sigma|\right],$$

where $A_f^H(r)$ denotes that A's random coins are fixed to the string r.

CMC and CC

cmc and CC

- How can we relate cmc to CC?

cmc and CC

- How can we relate cmc to CC?
  - Intuition: if A_f^H is computing $f_{G,H}$ (defined by DAG G and RO H), then the pebbling cost ($\mathsf{CC}(G)$) should translate to $\mathsf{cmc}(A_f)$.

cmc and CC

- How can we relate cmc to CC?
  - Intuition: if A_f^H is computing $f_{G,H}$ (defined by DAG G and RO H), then the pebbling cost ($\mathsf{CC}(G)$) should translate to $\mathsf{cmc}(A_f)$.

Theorem 5 (Alwen-Serbinenko 2015)

Let A_f^H be an algorithm computing a function $f_{G,H}$ defined by a DAG G and random oracle $H\colon \{0,1\}^* \to \{0,1\}^\lambda$.

cmc and CC

- How can we relate cmc to CC?
  - Intuition: if A_f^H is computing $f_{G,H}$ (defined by DAG G and RO H), then the pebbling cost ($\mathsf{CC}(G)$) should translate to $\mathsf{cmc}(A_f)$.

Theorem 5 (Alwen-Serbinenko 2015)

Let A_f^H be an algorithm computing a function $f_{G,H}$ defined by a DAG G and random oracle $H\colon \{0,1\}^* \to \{0,1\}^\lambda$. Then, in the parallel random oracle model (PROM), we have $\mathsf{cmc}(A_f) = \Omega(\lambda \cdot \mathsf{CC}(G))$.

cmc and CC

- How can we relate cmc to CC?
  - Intuition: if A_f^H is computing $f_{G,H}$ (defined by DAG G and RO H), then the pebbling cost ($\mathsf{CC}(G)$) should translate to $\mathsf{cmc}(A_f)$.

Theorem 5 (Alwen-Serbinenko 2015)

Let A_f^H be an algorithm computing a function $f_{G,H}$ defined by a DAG G and random oracle $H\colon \{0,1\}^* \to \{0,1\}^\lambda$. Then, in the parallel random oracle model (PROM), we have $\mathsf{cmc}(A_f) = \Omega(\lambda \cdot \mathsf{CC}(G))$.

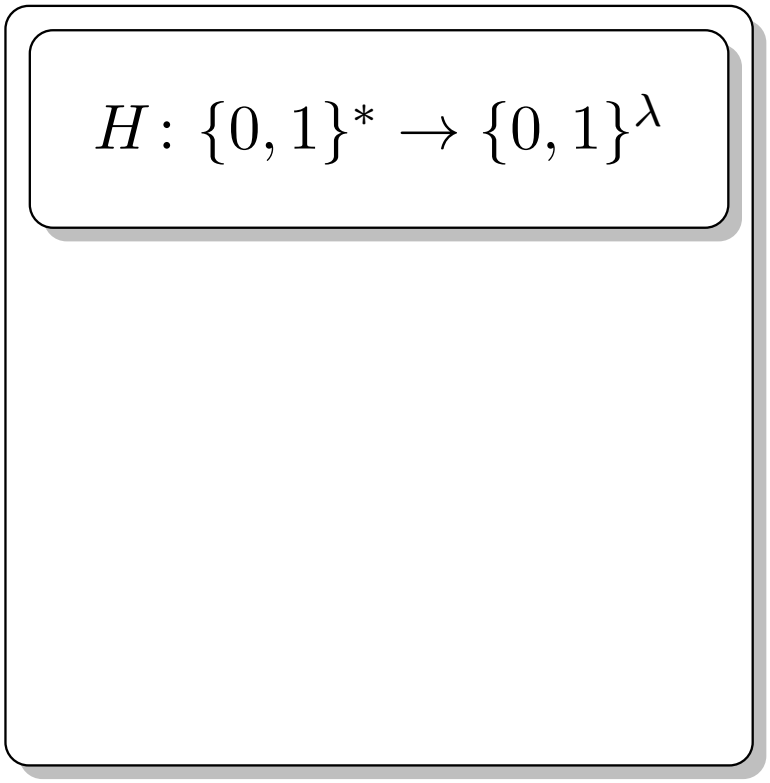
- We will give a very high-level overview of the proof.

Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.

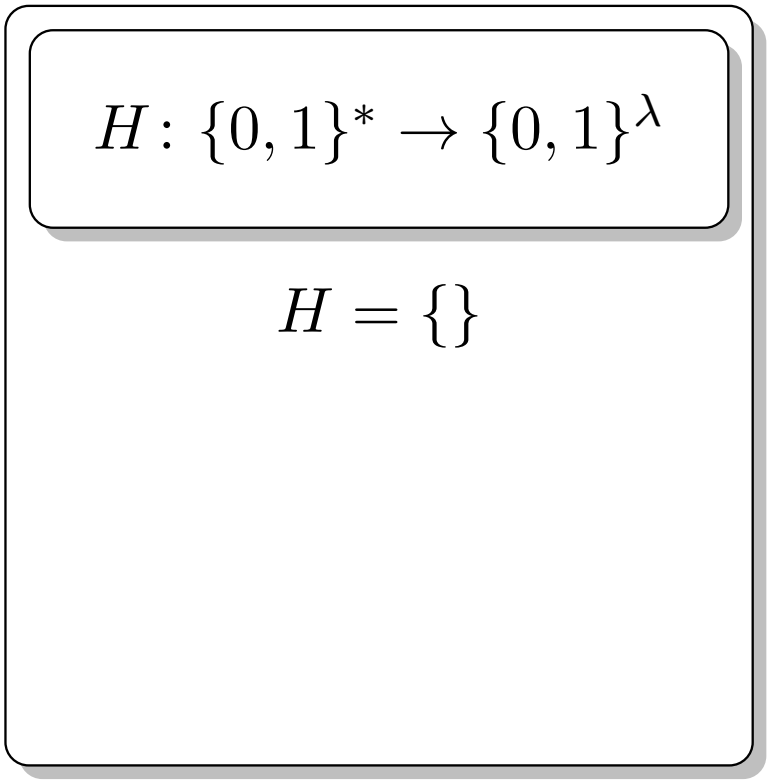
Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.



Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.



Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.

$H : \{0,1\}^* \to \{0,1\}^\lambda$

$H = \{\}$

$\underline{H(y):}$
If $H[y]$ undefined
$H[y] \xleftarrow{\$} \{0,1\}^n$
return $H[y]$

Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.

$H : \{0,1\}^* \to \{0,1\}^\lambda$

$H = \{\}$

$\underline{H(y):}$
If $H[y]$ undefined
$H[y] \xleftarrow{\$} \{0,1\}^n$
return $H[y]$

$\mathbb{U}_{\{0,1\}^\lambda}$

Review: Random Oracle Model

- Recall: a random oracle $H : \{0,1\}^* \to \{0,1\}^\lambda$ has uniformly random ouptuts.

$H : \{0,1\}^* \to \{0,1\}^\lambda$

$H = \{\}$

$\underline{H(y):}$
If $H[y]$ undefined
$H[y] \xleftarrow{\$} \{0,1\}^n$
return $H[y]$

$\equiv$ $\mathbb{U}_{\{0,1\}^\lambda}$

ROM: Prediction Game

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

A^H *wins the prediction game* if for all $i \in [k]$:

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

A^H *wins the prediction game* if for all $i \in [k]$: (1) $y_i = H(x_i)$, and

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

A^H *wins the prediction game* if for all $i \in [k]$: (1) $y_i = H(x_i)$, and (2) $x_i \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of queries made by A to H before outputting.

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

A^H *wins the prediction game* if for all $i \in [k]$: (1) $y_i = H(x_i)$, and (2) $x_i \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of queries made by A to H before outputting.

Fact 1

Any algorithm A^H wins the prediction game with probability at most $2^{-k \cdot \lambda}$ over the choice of H.

ROM: Prediction Game

- For a random oracle H and a probabilistic algorithm A^H (with access to H), we define the *prediction game* as follows.

Prediction Game

1. Output $(x_1, y_1), \ldots, (x_k, y_k) \leftarrow A^H$.

A^H *wins the prediction game* if for all $i \in [k]$: (1) $y_i = H(x_i)$, and (2) $x_i \notin \mathcal{Q}$, where $\mathcal{Q}$ is the set of queries made by A to H before outputting.

Fact 1

Any algorithm A^H wins the prediction game with probability at most $2^{-k \cdot \lambda}$ over the choice of H.

Fact 2 (ROs are Incompressible)

Any $A^H(h)$ given a hint $h \in \{0,1\}^s$ (which may depend on H) wins the prediction game with probability at most $2^{-k \cdot \lambda + s}$.

Parallel Random Oracle Model (PROM)

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.
  - $\mathbf{a}_1 = (H(x_1^{(1)}), \ldots, H(x_{r_1}^{(1)}))$.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.
  - $\mathbf{a}_1 = (H(x_1^{(1)}), \ldots, H(x_{r_1}^{(1)}))$.
    - Answers to RO queries.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.
  - $\mathbf{a}_1 = (H(x_1^{(1)}), \ldots, H(x_{r_1}^{(1)}))$.
    - Answers to RO queries.
  - $(\sigma_2, \mathbf{q}_2 = (x_1^{(2)}, \ldots, x_{r_2}^{(2)})) \leftarrow A(\sigma_1, \mathbf{a}_1)$.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.
  - $\mathbf{a}_1 = (H(x_1^{(1)}), \ldots, H(x_{r_1}^{(1)}))$.
    - Answers to RO queries.
  - $(\sigma_2, \mathbf{q}_2 = (x_1^{(2)}, \ldots, x_{r_2}^{(2)})) \leftarrow A(\sigma_1, \mathbf{a}_1)$.
  - $\ldots$
  - $(\sigma_i, \mathbf{q}_i = (x_1^{(i)}, \ldots, x_{r_i}^{(i)})) \leftarrow A(\sigma_{i-1}, \mathbf{a}_{i-1})$.

Parallel Random Oracle Model (PROM)

- Extends ROM to allow adversary to make *parallel queries*.
- Let $A(x)$ be a PROM algorithm with input x and access to RO H; it executes as follows.
  - **Initial Input/State**: $\sigma_0 = x$.
  - $(\sigma_1, \mathbf{q}_1 = (x_1^{(1)}, \ldots, x_{r_1}^{(1)})) \leftarrow A(\sigma_0)$.
    - New state + batch of RO queries.
  - $\mathbf{a}_1 = (H(x_1^{(1)}), \ldots, H(x_{r_1}^{(1)}))$.
    - Answers to RO queries.
  - $(\sigma_2, \mathbf{q}_2 = (x_1^{(2)}, \ldots, x_{r_2}^{(2)})) \leftarrow A(\sigma_1, \mathbf{a}_1)$.
  - $\cdots$
  - $(\sigma_i, \mathbf{q}_i = (x_1^{(i)}, \ldots, x_{r_i}^{(i)})) \leftarrow A(\sigma_{i-1}, \mathbf{a}_{i-1})$.
  - $\cdots$
  - $y \leftarrow A(\sigma_t, \mathbf{a}_t)$

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\mathrm{Trace}_{A,H}(x) = \{\!(\sigma_i, \mathbf{q}_i, \mathbf{a}_i)\!\}_{i=1}^{t}.$$

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i=1}^{t}.$$

  - Allows us to define the cmc of an execution trace:

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i=1}^{t}.$$

  - Allows us to define the cmc of an execution trace:

$$\mathsf{cmc}(\text{Trace}_{A,H}(x)) = \sum_{i=1}^{t}(|\sigma_i| + |\mathbf{a}_i|).$$

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i=1}^{t}.$$

  - Allows us to define the cmc of an execution trace:

$$\mathsf{cmc}(\text{Trace}_{A,H}(x)) = \sum_{i=1}^{t}(|\sigma_i| + |\mathbf{a}_i|).$$

  - Finally, suppose that $A(x)$ is computing $f_{G,H}(x)$.

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i=1}^{t}.$$

  - Allows us to define the cmc of an execution trace:

$$\mathsf{cmc}(\text{Trace}_{A,H}(x)) = \sum_{i=1}^{t}(|\sigma_i| + |\mathbf{a}_i|).$$

  - Finally, suppose that $A(x)$ is computing $f_{G,H}(x)$. Then, we can define the cmc of $f_{G,H}$ as

Parallel Random Oracle Model (PROM)

- PROM Algorithm $A(x)$
  - Fixing A, x, and RO H, we can define the *execution trace* as

$$\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i=1}^{t}.$$

  - Allows us to define the cmc of an execution trace:

$$\mathsf{cmc}(\text{Trace}_{A,H}(x)) = \sum_{i=1}^{t}(|\sigma_i| + |\mathbf{a}_i|).$$

  - Finally, suppose that $A(x)$ is computing $f_{G,H}(x)$. Then, we can define the cmc of $f_{G,H}$ as

$$\mathsf{cmc}(f_{G,H}) = \min_{A,x} \mathop{\mathbb{E}}_{H} \left[\mathsf{cmc}(\text{Trace}_{A,H}(x))\right].$$

Towards the Proof: Label Distinctness

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.
  - Define labels $L_1 = H(L_0, 0^{\lambda})$, $L_2 = H(L_1, 0^{\lambda})$, and

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.
  - Define labels $L_1 = H(L_0, 0^{\lambda})$, $L_2 = H(L_1, 0^{\lambda})$, and

$$L_3 = H(L_2, L_1)$$

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.
  - Define labels $L_1 = H(L_0, 0^{\lambda})$, $L_2 = H(L_1, 0^{\lambda})$, and

$$L_3 = H(L_2, L_1)$$

$$\ldots$$

$$L_v = H(L_{v-1}, L_{r(v)})$$

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.
  - Define labels $L_1 = H(L_0, 0^{\lambda})$, $L_2 = H(L_1, 0^{\lambda})$, and

$$L_3 = H(L_2, L_1)$$
$$\ldots$$
$$L_v = H(L_{v-1}, L_{r(v)})$$
$$\ldots$$
$$L_n = H(L_{n-1}, L_{r(n)}).$$

Towards the Proof: Label Distinctness

- Suppose we have a random oracle $H\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$.
- Suppose we are given a DAG G on n vertices $V = \{1, \ldots, n\}$ such that $\delta(G) = 2$.
- Moreover, suppose that each vertex v has parents v and $r(v) < v - 1$ for some function r.
- Define $x = L_0 \in \{0,1\}^{\lambda}$ as the initial input.
  - Define labels $L_1 = H(L_0, 0^{\lambda})$, $L_2 = H(L_1, 0^{\lambda})$, and

$$L_3 = H(L_2, L_1)$$
$$\ldots$$
$$L_v = H(L_{v-1}, L_{r(v)})$$
$$\ldots$$
$$L_n = H(L_{n-1}, L_{r(n)}).$$

- **Question**: what is the probability that two labels collide?

Label Distinctness

- **Question**: what is the probability that two labels collide?

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{\lambda}}$$

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{+\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!
    - $\Pr[\overline{E}_2 \mid E_1] \leqslant 2^{-\lambda}$. $\overline{E_2} = \{L_1 = L_2\}$

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!
    - $\Pr[\overline{E}_2 \mid E_1] \leqslant 2^{-\lambda}$.
  - Inductive case:

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!
    - $\Pr[\overline{E}_2 \mid E_1] \leqslant 2^{-\lambda}$.
  - Inductive case:
    - $\Pr[\overline{E}_i | E_{i-1}] = \Pr[H(L_i, L_{r(i)}) \in \{L_1, \ldots, L_{i-1}\} \mid E_{i-1}] \leqslant (i-1) \cdot 2^{-\lambda}$.

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!
    - $\Pr[\overline{E}_2 \mid E_1] \leqslant 2^{-\lambda}$.
  - Inductive case:
    - $\Pr[\overline{E}_i | E_{i-1}] = \Pr[H(L_i, L_{r(i)}) \in \{L_1, \ldots, L_{i-1}\} \mid E_{i-1}] \leqslant (i-1) \cdot 2^{-\lambda}$.
  - Union bound:

Label Distinctness

- **Question**: what is the probability that two labels collide?

Claim 2

$$\Pr_H[\exists v \neq v' : L_v = L_{v'}] \leqslant \frac{n^2}{2^{-\lambda}}$$

- Proof intuition: induction + collision-resistance of Random Oracles.
- Let E_i be the event that labels $L_1, \ldots, L_i$ are all distinct.
  - Notice that $\Pr[E_1] = 1$ (there are no other labels).
  - What is the probability L_2 and L_1 are distinct?
    - Note that L_1 is fixed when L_2 is computed!
    - $\Pr[\overline{E}_2 \mid E_1] \leqslant 2^{-\lambda}$.
  - Inductive case:
    - $\Pr[\overline{E}_i | E_{i-1}] = \Pr[H(L_i, L_{r(i)}) \in \{L_1, \ldots, L_{i-1}\} \mid E_{i-1}] \leqslant (i-1) \cdot 2^{-\lambda}$.
  - Union bound:
    - $\Pr[\overline{E}_n] \leqslant \sum_{i=1}^{n} (i-1) \cdot 2^{-\lambda} \leqslant n^2 \cdot 2^{-\lambda}$.

Pre-labeling and Label Collisions

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\textbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \textbf{prelab}(v)$ for some node v?

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\textbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \textbf{prelab}(v)$ for some node v?

- **Answer**: probability at most $n \cdot q \cdot 2^{-\lambda}$.

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \mathbf{prelab}(v)$ for some node v?

- **Answer**: probability at most $n \cdot q \cdot 2^{-\lambda}$.
  - Let z_i be the i-th query to the RO such that $z_i \neq \mathbf{prelab}(v)$ for any $v \leqslant n$.

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \mathbf{prelab}(v)$ for some node v?

- **Answer**: probability at most $n \cdot q \cdot 2^{-\lambda}$.
  - Let z_i be the i-th query to the RO such that $z_i \neq \mathbf{prelab}(v)$ for any $v \leqslant n$.
  - Then, we have

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \mathbf{prelab}(v)$ for some node v?

- **Answer**: probability at most $n \cdot q \cdot 2^{-\lambda}$.
  - Let z_i be the i-th query to the RO such that $z_i \neq \mathbf{prelab}(v)$ for any $v \leqslant n$.
  - Then, we have

$$\Pr[H(z_i) \in \{L_1, \ldots, L_n\}] \leqslant n \cdot 2^{-\lambda}$$

Pre-labeling and Label Collisions

- Define the *pre-labeling* of a node v as $\mathbf{prelab}(v) = (L_{v-1}, L_{r(v)})$.

- **Question**: suppose we can make at most q queries to the random oracle.
  - What is the probability we find z such that $L_v = H(z)$ but $z \neq \mathbf{prelab}(v)$ for some node v?

- **Answer**: probability at most $n \cdot q \cdot 2^{-\lambda}$.
  - Let z_i be the i-th query to the RO such that $z_i \neq \mathbf{prelab}(v)$ for any $v \leqslant n$.
  - Then, we have

$$\Pr[H(z_i) \in \{L_1, \ldots, L_n\}] \leqslant n \cdot 2^{-\lambda}$$
$$\Pr[\exists i \leqslant q \colon H(z_i) \in \{L_1, \ldots, L_n\}] \leqslant n \cdot q \cdot 2^{-\lambda}.$$

Ex Post Facto Pebbling

Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.

Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.
- Again, fix PROM algorithm A, input x, and RO H and recall the execution trace $\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i \in [t]}$.

Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.
- Again, fix PROM algorithm A, input x, and RO H and recall the execution trace $\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i \in [t]}$.
- For each node $v \in [n]$, track L_v as follows.

Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.
- Again, fix PROM algorithm A, input x, and RO H and recall the execution trace $\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i \in [t]}$.
- For each node $v \in [n]$, track L_v as follows.
  - Note the rounds $i \in [t]$ where L_v appears as the *input* to the RO query (e.g., $L_v \in \mathbf{q}_i$).

Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.

- Again, fix PROM algorithm A, input x, and RO H and recall the execution trace $\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i \in [t]}$.

- For each node $v \in [n]$, track L_v as follows.
  - Note the rounds $i \in [t]$ where L_v appears as the *input* to the RO query (e.g., $L_v \in \mathbf{q}_i$).
  - Note the rounds $i \in [t]$ where L_v appears at the *output* of the RO query (e.g., $L_v \in \mathbf{a}_i$).

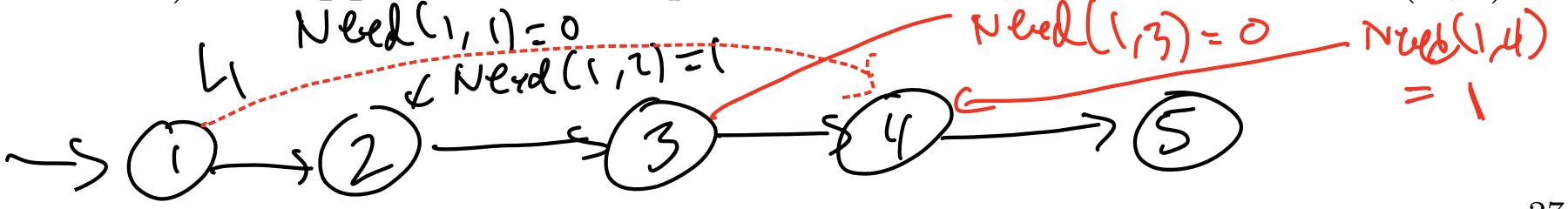
Ex Post Facto Pebbling

- We now describe the *ex post facto pebbling* strategy.

- Again, fix PROM algorithm A, input x, and RO H and recall the execution trace $\text{Trace}_{A,H}(x) = \{\sigma_i, \mathbf{q}_i, \mathbf{a}_i\}_{i \in [t]}$.

- For each node $v \in [n]$, track L_v as follows.
  - Note the rounds $i \in [t]$ where L_v appears as the *input* to the RO query (e.g., $L_v \in \mathbf{q}_i$).
  - Note the rounds $i \in [t]$ where L_v appears at the *output* of the RO query (e.g., $L_v \in \mathbf{a}_i$).

- Define indicator $\text{Need}(v, i) = 1$ if and only if the next time (after round i) L_v appears is an *input* to the RO; otherwise $\text{Need}(v, i) = 0$.



Ex Post Facto Pebbling

Ex Post Facto Pebbling

- Define the ex post facto pebbling as

Ex Post Facto Pebbling

- Define the ex post facto pebbling as

$$\mathbf{P} = \{P_i = \{v \colon \text{Need}(v, i) = 1\}\}_{i=1}^{t}.$$

Ex Post Facto Pebbling

- Define the ex post facto pebbling as

$$\mathbf{P} = \{P_i = \{v \colon \text{Need}(v, i) = 1\}\}_{i=1}^{t}.$$

- **Missing pieces**:

Ex Post Facto Pebbling

- Define the ex post facto pebbling as

$$\mathbf{P} = \{P_i = \{v \colon \text{Need}(v, i) = 1\}\}_{i=1}^{t}.$$

- **Missing pieces**:
  1. $\mathbf{P}$ is a valid pebbling of G, and

Ex Post Facto Pebbling

- Define the ex post facto pebbling as

$$\mathbf{P} = \{P_i = \{v \colon \mathrm{Need}(v, i) = 1\}\}_{i=1}^{t}.$$

- **Missing pieces**:
  1. $\mathbf{P}$ is a valid pebbling of G, and
  2. $\mathsf{CC}(\mathbf{P}) \geqslant \mathsf{CC}(G)$.

↑

Relate $|P_i|$ to $|O_i| + |q_i|$

Next Time: wrap up the proof + PQ Crypto Start