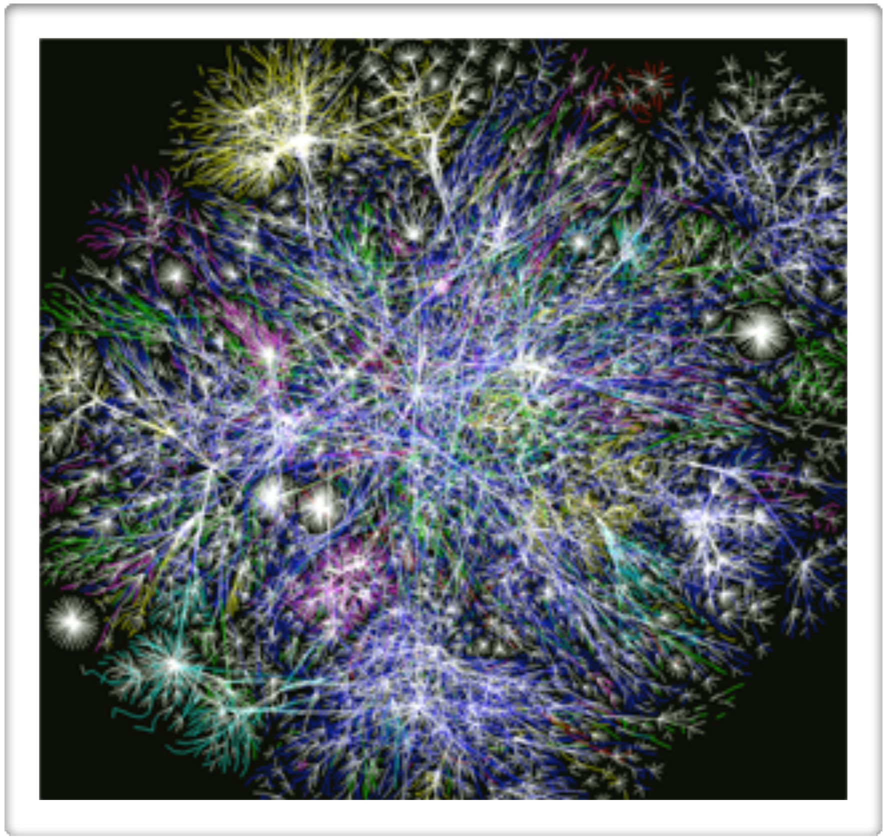
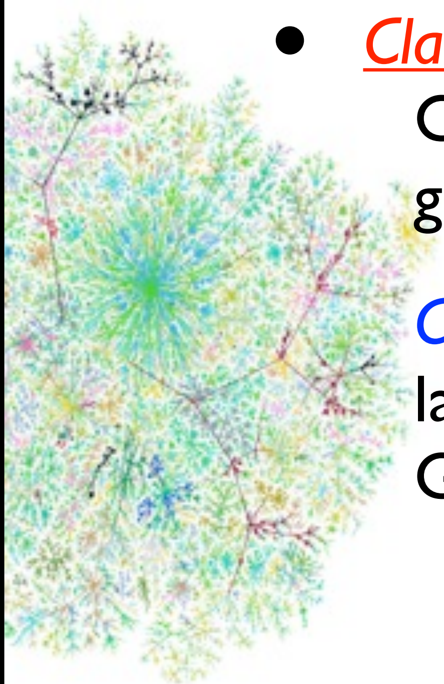
Graph Synopses, Sketches, *and* Streams: A Survey



Sudipto Guha
University of Pennsylvania

Andrew McGregor
University of Massachusetts

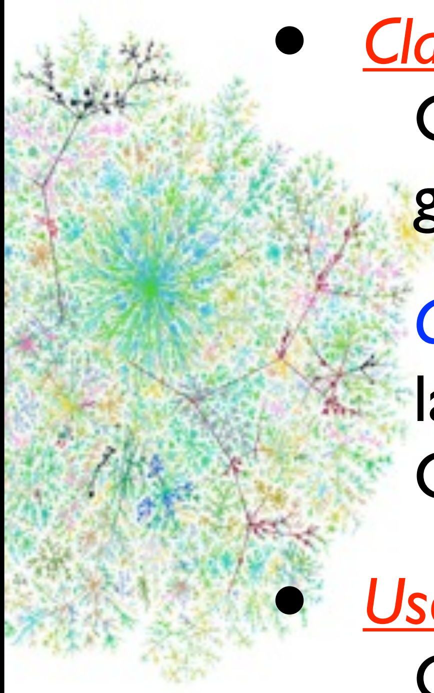
Massive Graphs



- *Classic Big Graphs:*
  Call graph (5×10^8 nodes), web graph (5×10^{10} nodes), IP graph (2^{32} nodes), social networks (10^9 nodes), ...

  Challenge: Can't use conventional algorithms on graphs this large. Sometimes can't even store graph in memory! Graphs may be dynamic and/or distributed.

Massive Graphs

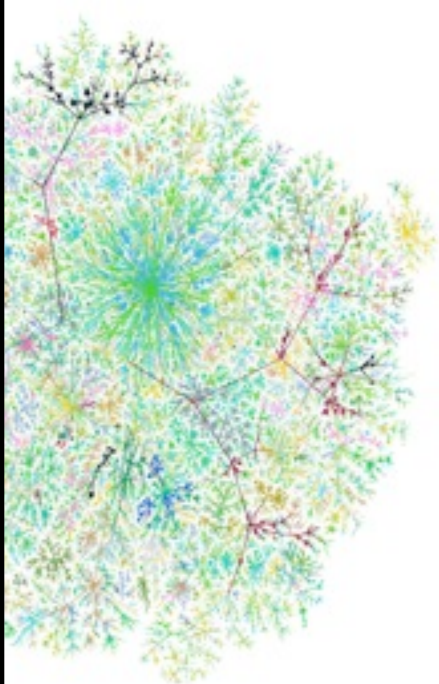


- *Classic Big Graphs:*
  Call graph (5×10^8 nodes), web graph (5×10^{10} nodes), IP graph (2^{32} nodes), social networks (10^9 nodes), ...

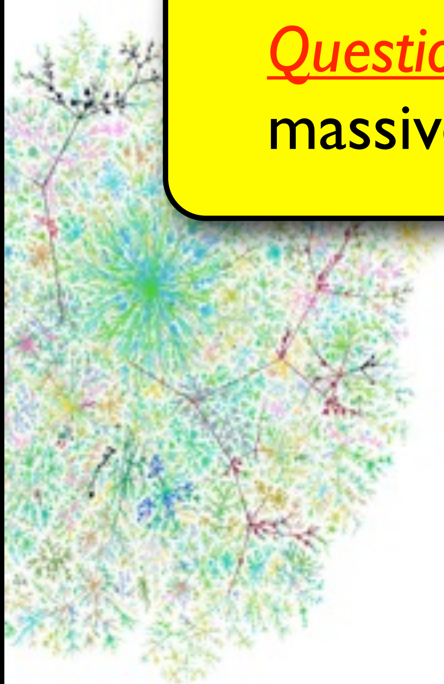
  Challenge: Can't use conventional algorithms on graphs this large. Sometimes can't even store graph in memory! Graphs may be dynamic and/or distributed.

- *Use Abstraction of Structure:*
  Graphs are a natural way to encode structural information where we have data about both basic entities and their relationships. Examples include graphical networks, citation networks, protein interaction and metabolic networks, ...

Focus of Tutorial



Focus of Tutorial



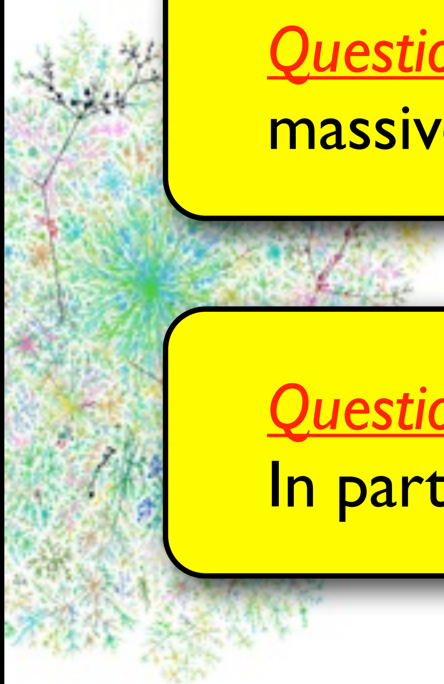
Question 1: What are appropriate synopsis data structures for massive graphs? How do we trade-off space and accuracy?

Focus of Tutorial

Question 1: What are appropriate synopsis data structures for massive graphs? How do we trade-off space and accuracy?

Question 2: How can we construct these synopses efficiently? In particular, what is the input is streaming or distributed?

Focus of Tutorial



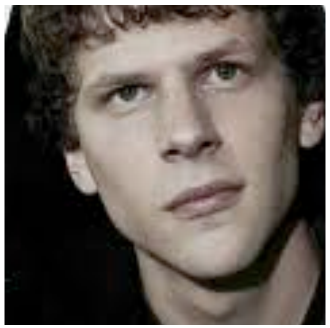
Question 1: What are appropriate synopsis data structures for massive graphs? How do we trade-off space and accuracy?

Question 2: How can we construct these synopses efficiently? In particular, what is the input is streaming or distributed?

- Tutorial focuses on the algorithmic and theoretical issues. Consider arbitrary graphs rather than being domain specific.

  This Talk: Definitions & Basic Building Blocks

  Next Talk: Applications & Extensions



[Mark]() and [Erica]() are now friends.

Like · Add Friend

Mark and Erica are no longer friends.

 Like · Add Friend

Eduardo and Mark are now friends.

Like · Add Friend

Tyler and Cameron are friends with Mark.

Like · Add Friend

Sean and Mark are now friends.

Like · Add Friend

Eduardo and Mark are no longer friends.

Like · Add Friend

Tyler and Cameron are no longer friends with Mark.

Like · Add Friend

Lawyers are now friends with everyone.

Like · Add Friend

Data Streams

- ***Input:*** Observe stream of edges on n nodes added/deleted.

Data Streams

- *Input:* Observe stream of edges on n nodes added/deleted.
- *Example:* Using $\tilde{O}(n)$ space, maintain connected components.

Data Streams

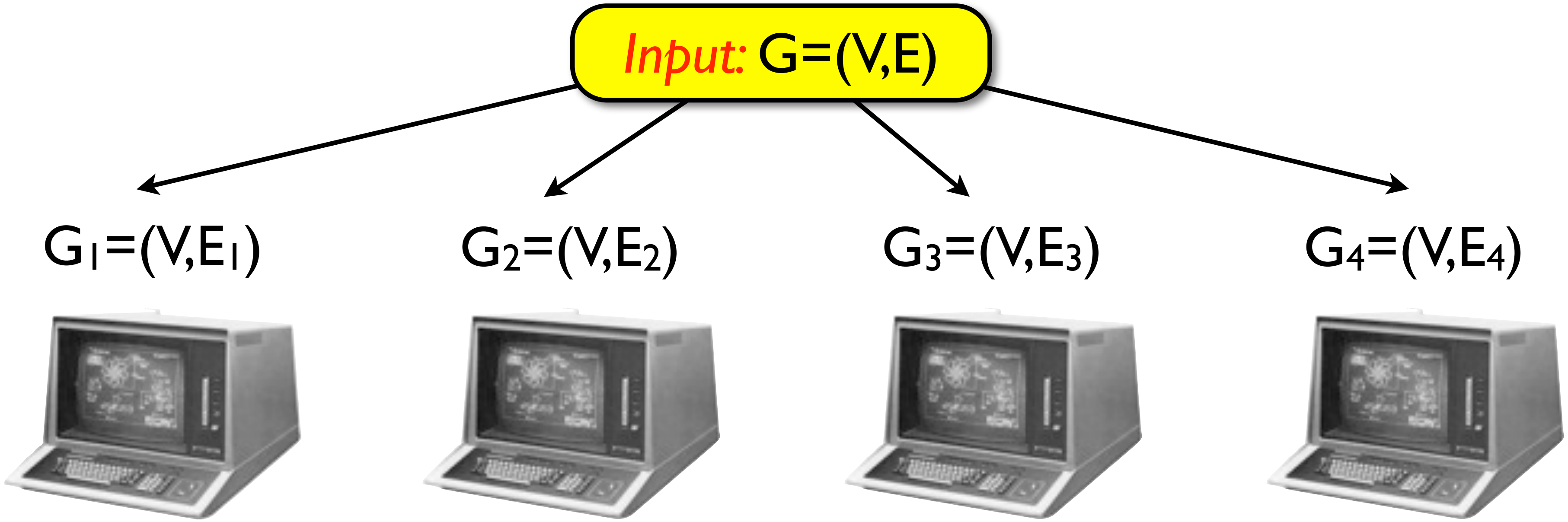
- *Input:* Observe stream of edges on n nodes added/deleted.
- *Example:* Using $\tilde{O}(n)$ space, maintain connected components.
- *Other Results:* Dense subgraphs, matchings, distances, clustering, partitioning and cuts, diameter, random walks, ...

e.g., [Feigenbaum, Kannan, McGregor, Suri, Zhang 2004, 2005], [McGregor 2005]
[Jowhari, Ghodsi 2005], [Zelke 2008], [Sarma, Gollapudi, Panigrahy 2008, 2009]
[Eggert, Kliemann, Srivastav 2009], [Epstein, Levin, Mestre, Segev 2009]
[Ahn, Guha 2009, 2011], [Kelner, Levine 2011], [Goel, Kapralov, Khanna 2012]

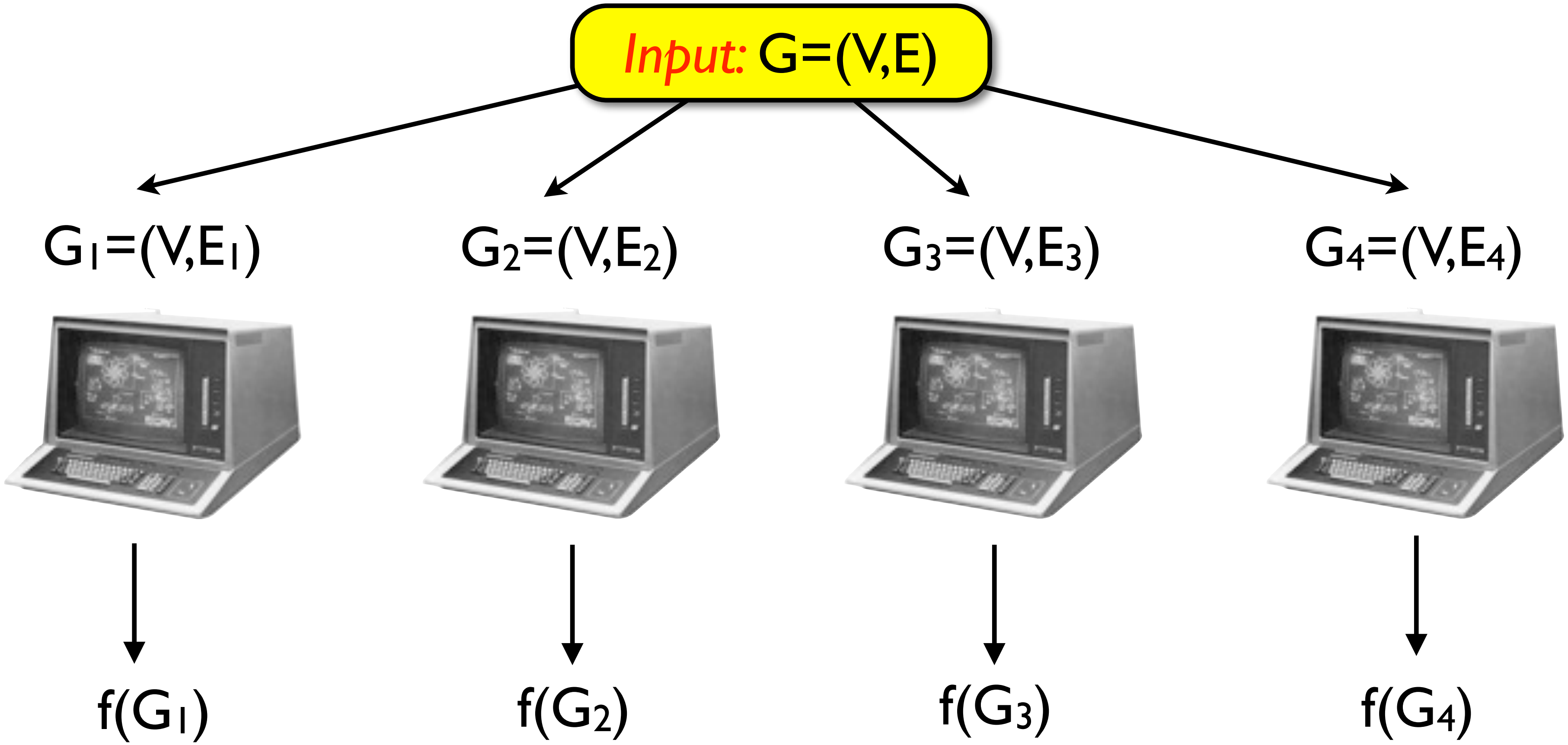
Distributed Processing

Input: G=(V,E)

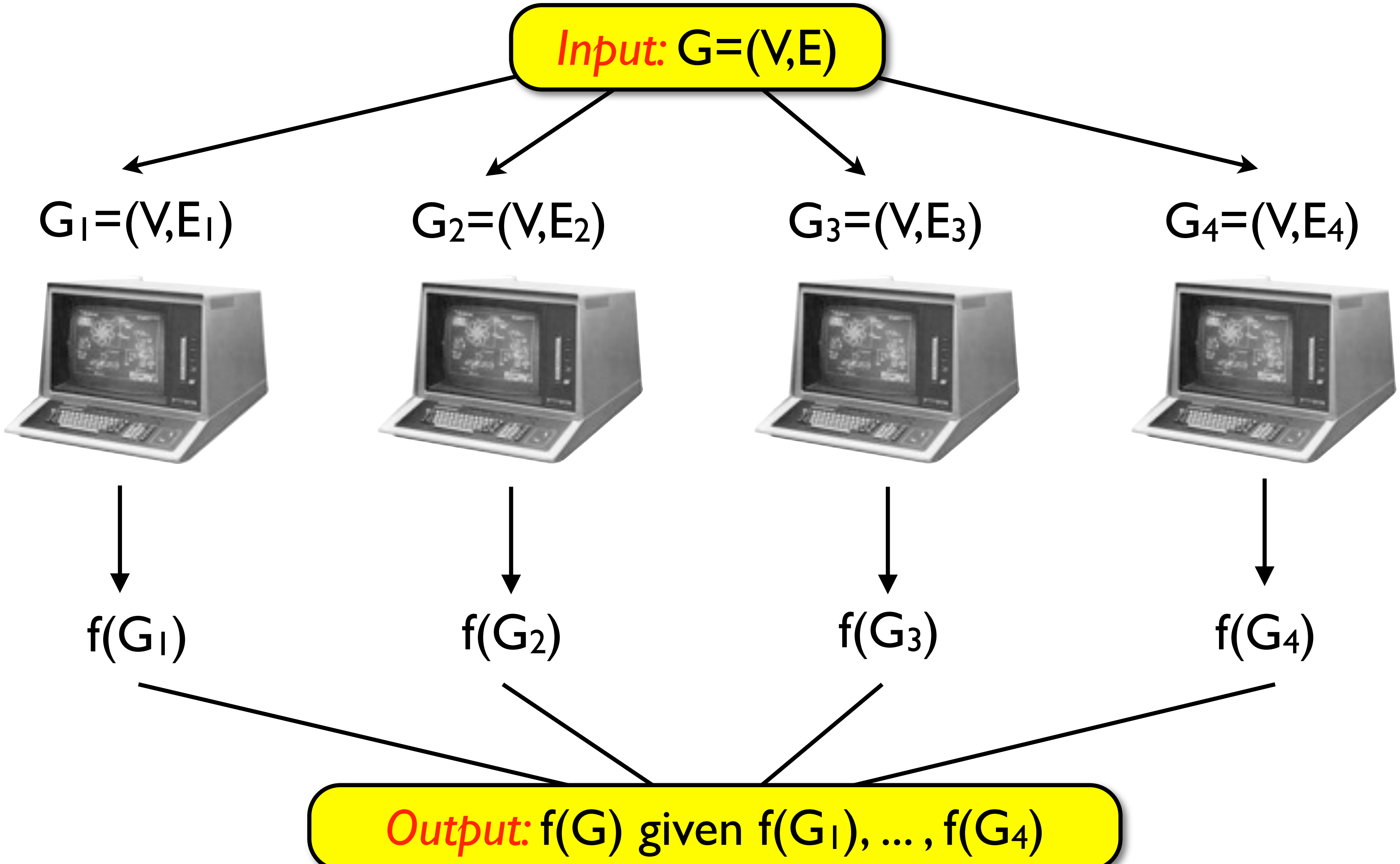
Distributed Processing



Distributed Processing

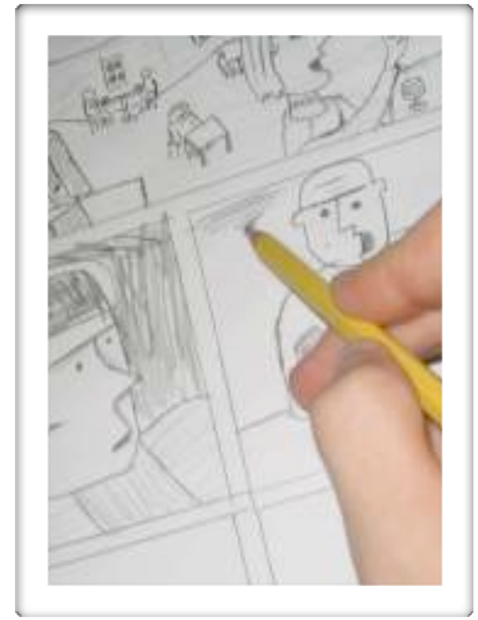


Distributed Processing



I. **Spanners**

II. **Sparsifiers**



III. **Sketches**

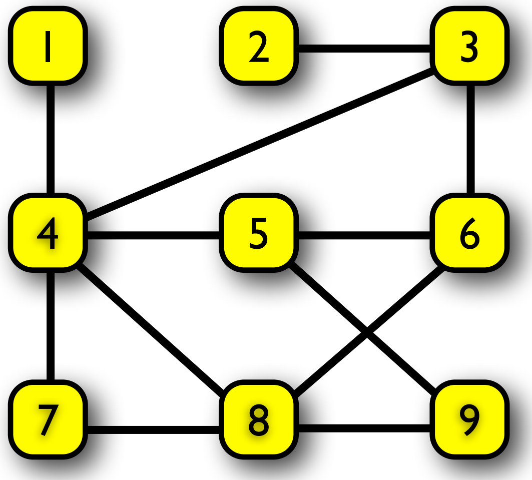
1. **Spanners**

Synopsis for Distance Estimation
"Greedy" Stream Algorithm
Extensions

Spanners & Distances

Spanners & Distances

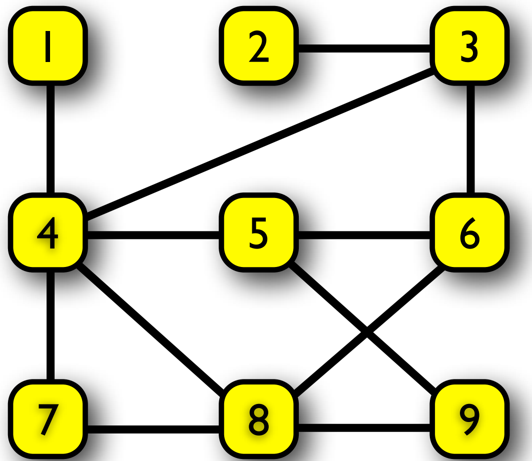
- *Measure:* The distance $d_G(u,v)$ between two nodes u, v is the length of the shortest path between the nodes.



Original Graph G

Spanners & Distances

- *Measure:* The distance $d_G(u,v)$ between two nodes u, v is the length of the shortest path between the nodes.



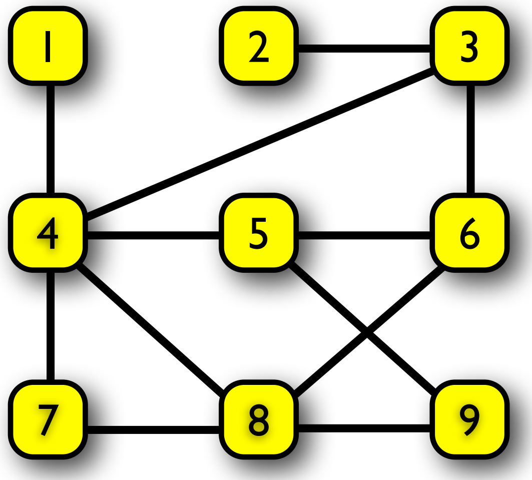
Original Graph G

- *Synopsis:* A subgraph H of G is a *k-spanner* if

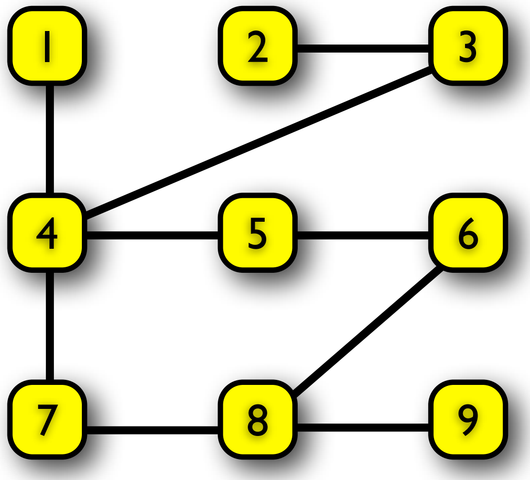
$$d_G(u,v) \leq d_H(u,v) \leq k\, d_G(u,v) \quad \text{for all node pairs.}$$

Spanners & Distances

- *Measure:* The distance $d_G(u,v)$ between two nodes u, v is the length of the shortest path between the nodes.



Original Graph G



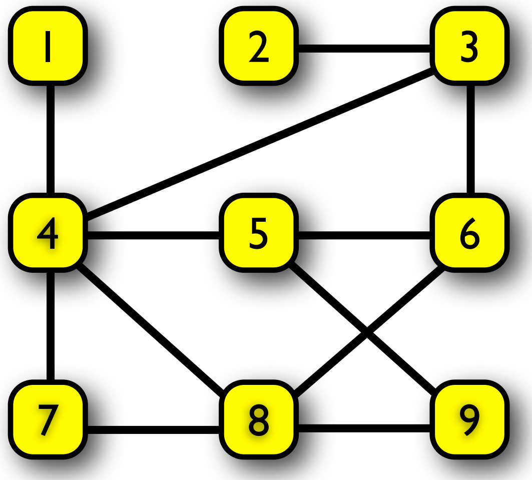
Spanner Graph H

- *Synopsis:* A subgraph H of G is a *k-spanner* if

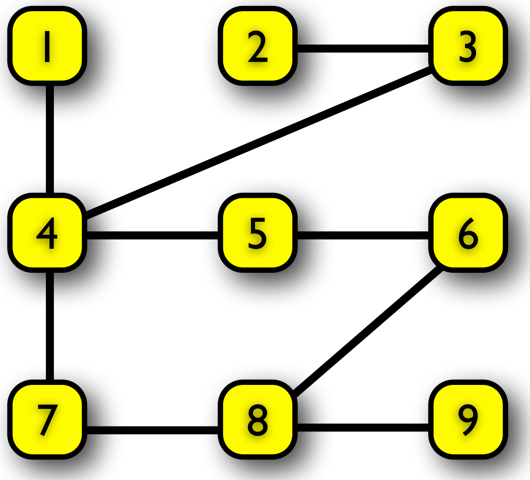
$$d_G(u,v) \leq d_H(u,v) \leq k\, d_G(u,v) \quad \text{for all node pairs.}$$

Spanners & Distances

- *Measure:* The distance $d_G(u,v)$ between two nodes u, v is the length of the shortest path between the nodes.



Original Graph G



Spanner Graph H

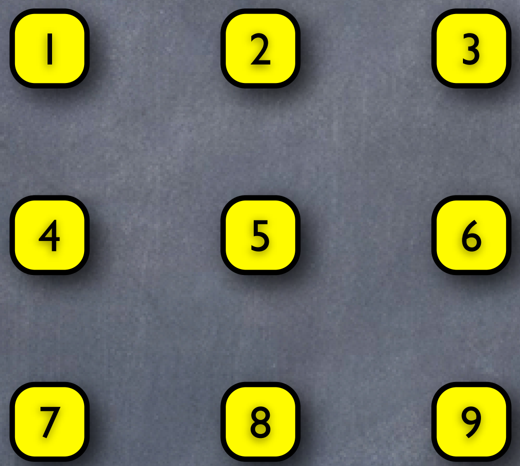
- *Synopsis:* A subgraph H of G is a *k-spanner* if

  $d_G(u,v) \leq d_H(u,v) \leq k\, d_G(u,v)$ for all node pairs.

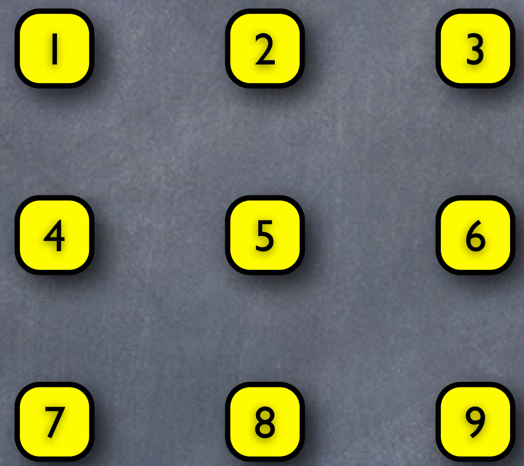
- *Thm:* Streaming construction using $O(n^{1+2/(k+1)})$ space.

Spanner: Algorithm

Spanner: Algorithm

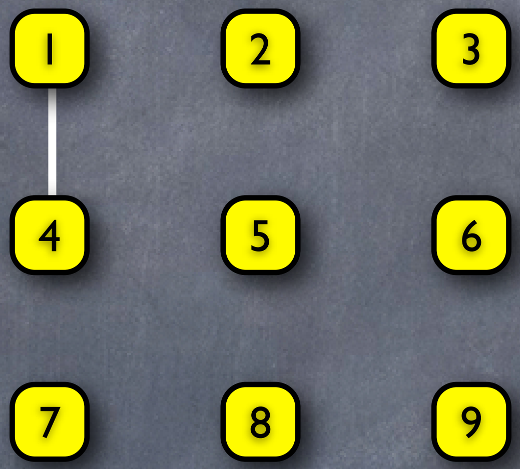


Spanner: Algorithm



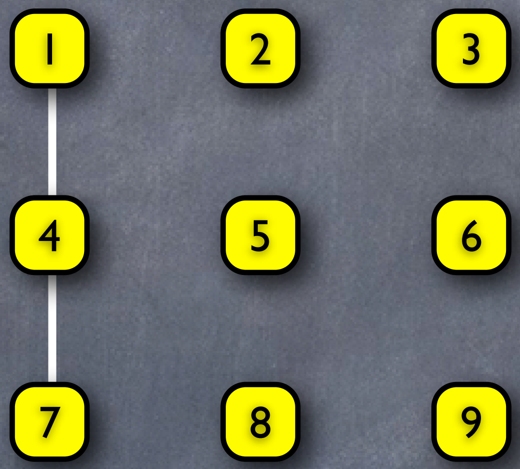
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



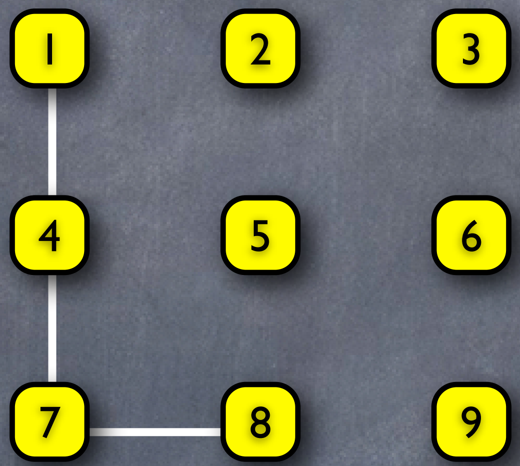
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



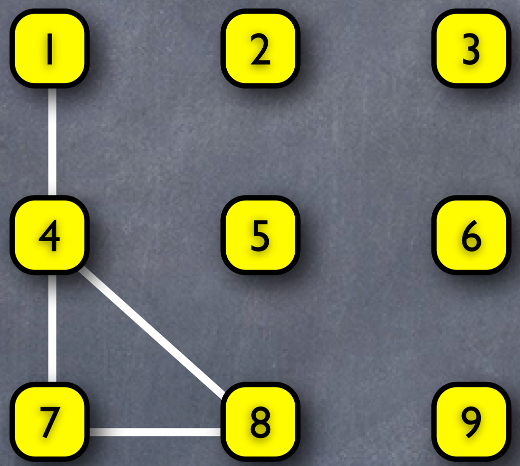
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



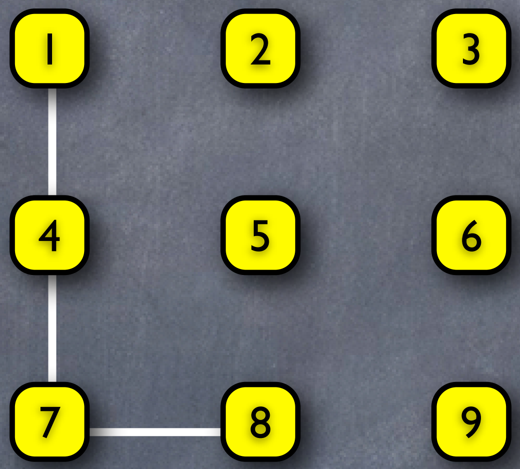
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



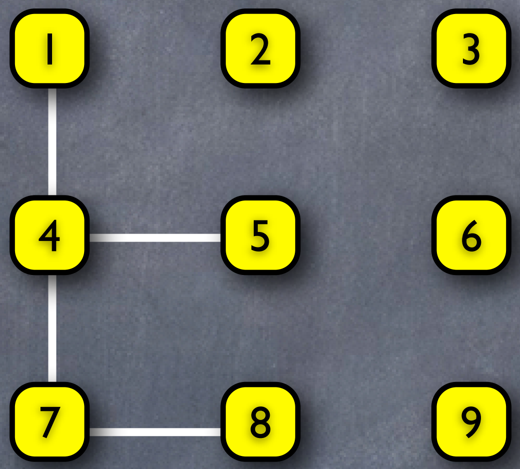
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



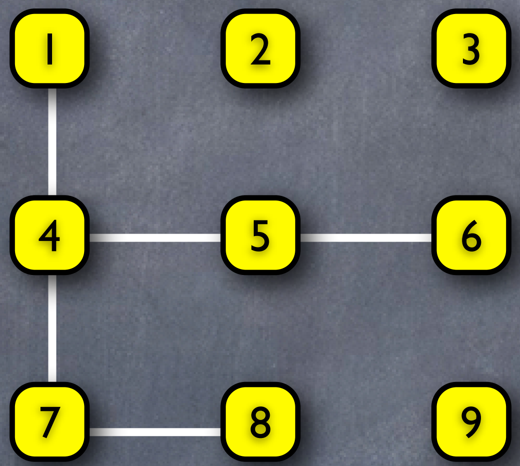
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



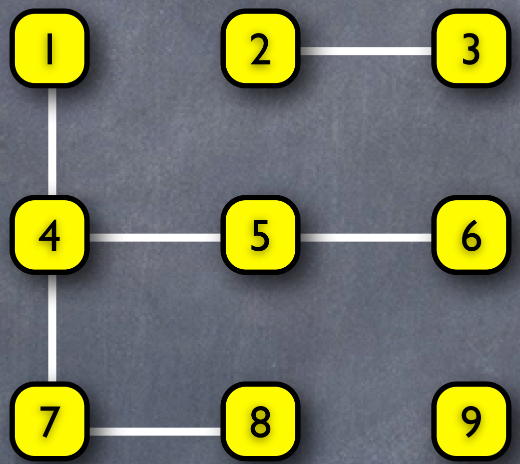
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



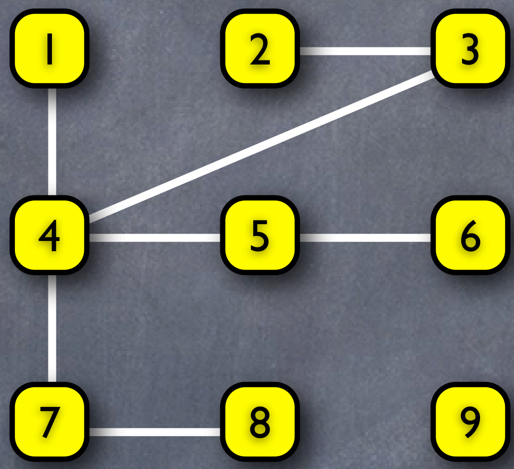
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



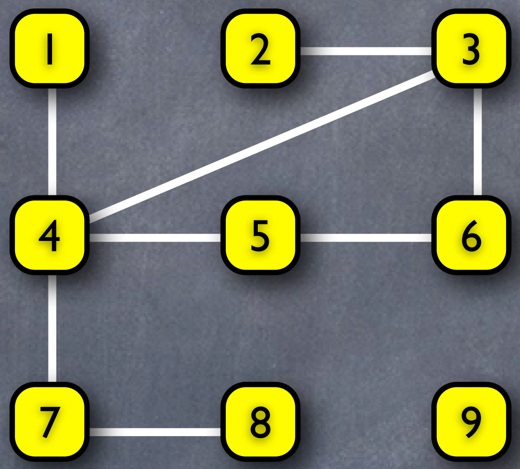
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



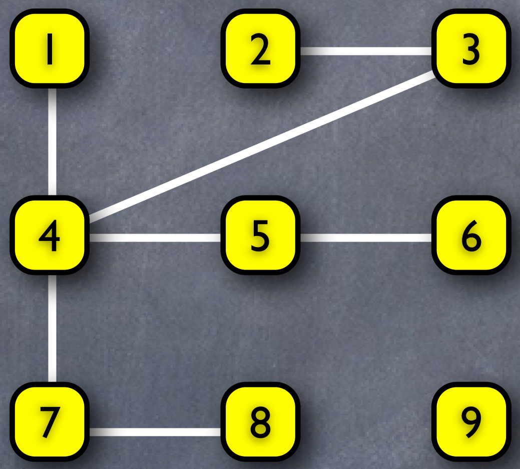
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



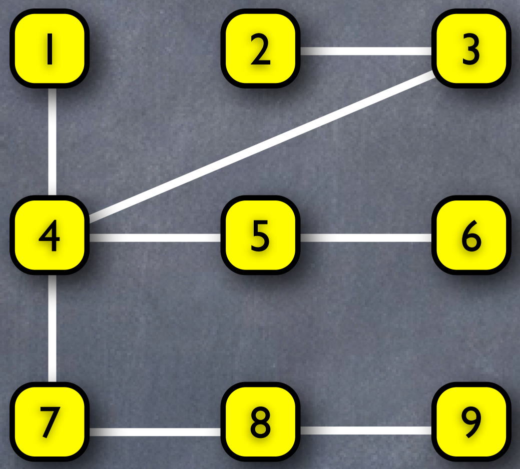
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



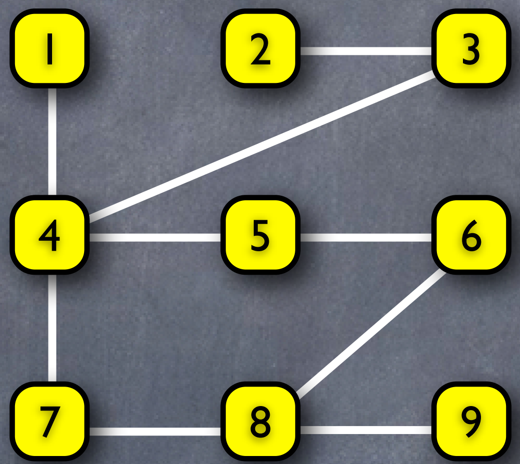
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



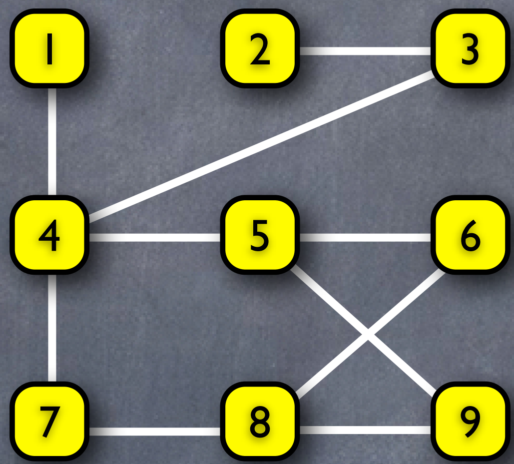
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



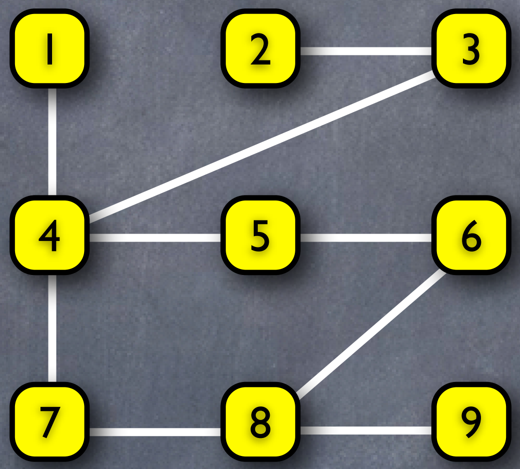
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



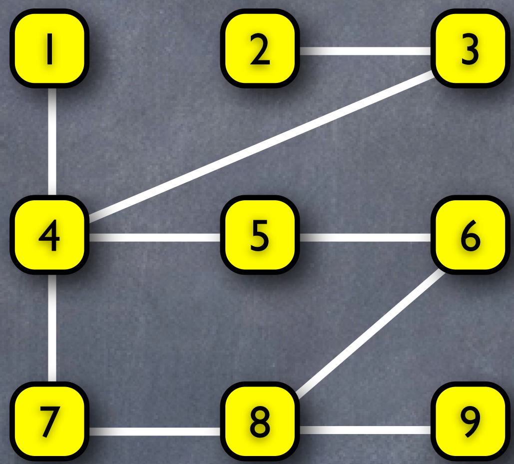
- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



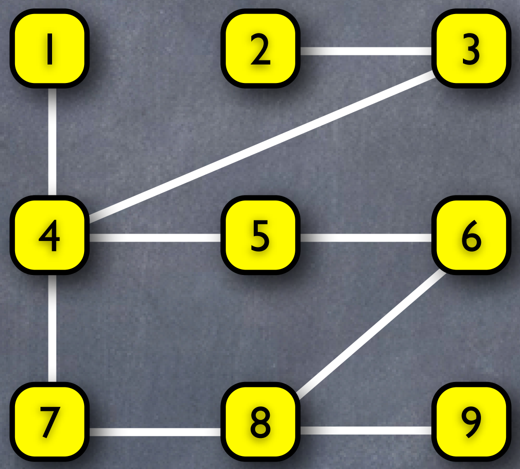
- **Algorithm:** Add new edge (u,v) to H if $d_H(u,v)>3$.

Spanner: Algorithm



- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.
- Lemma: All distances preserved up to a factor 3.

Spanner: Algorithm



- Algorithm: Add new edge (u,v) to H if $d_H(u,v)>3$.
- Lemma: All distances preserved up to a factor 3.
- Lemma: $O(n^{3/2})$ edges stored since shortest cycle among stored edges has length at least 5.

Spanners: Analysis

Spanners: Analysis

- If H has m edges, average degree is $d=2m/n$.

Spanners: Analysis

- If H has m edges, average degree is $d=2m/n$.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least $d'=d/2$

Spanners: Analysis

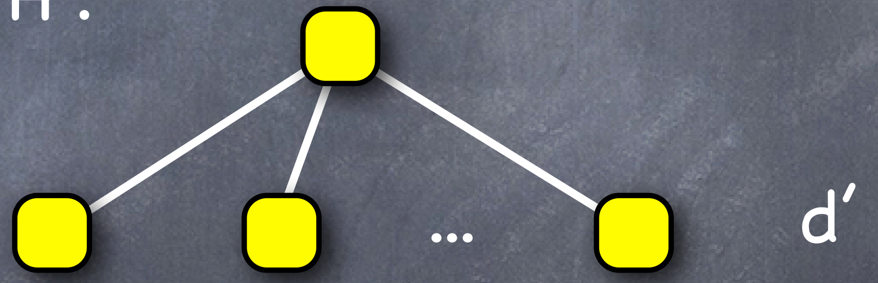
- If H has m edges, average degree is $d=2m/n$.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least $d'=d/2$
- Proof: Remove all nodes with degree $< d'$. Can only remove $< nd'=nd/2=m$ edges so H′ non-empty.

Spanners: Analysis

- If H has m edges, average degree is d=2m/n.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least d′=d/2
- Proof: Remove all nodes with degree < d′. Can only remove < nd′=nd/2=m edges so H′ non-empty.
- Consider node in H′:

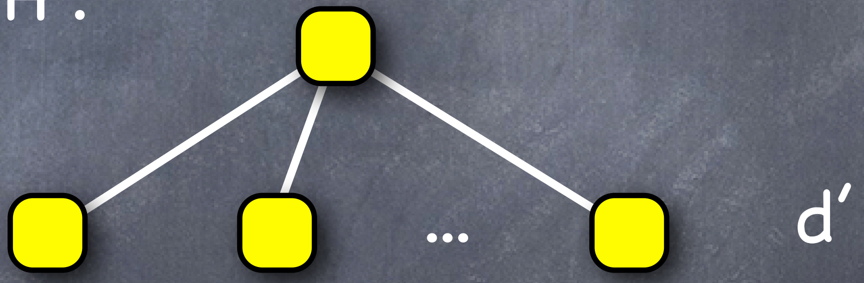
Spanners: Analysis

- If H has m edges, average degree is d=2m/n.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least d′=d/2
- Proof: Remove all nodes with degree < d′. Can only remove < nd′=nd/2=m edges so H′ non-empty.
- Consider node in H′:



Spanners: Analysis

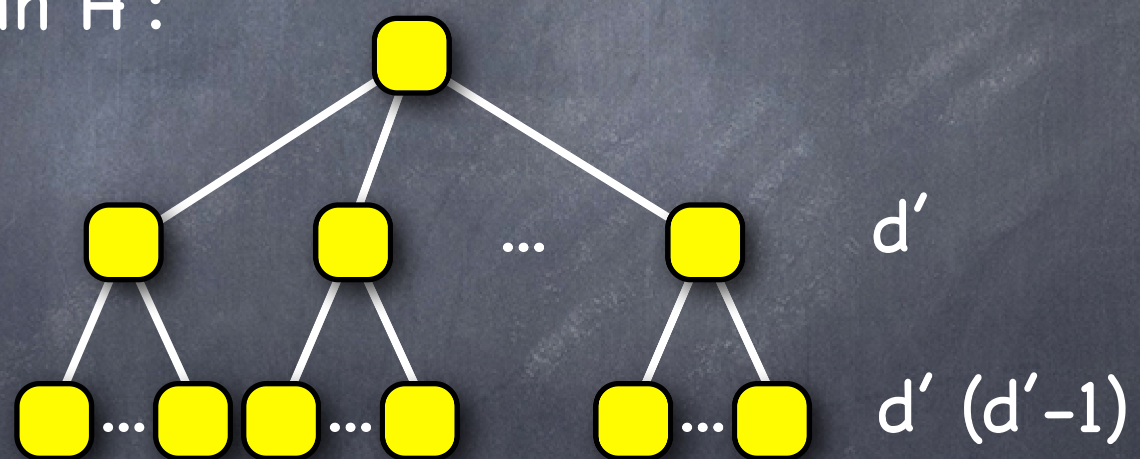
- If H has m edges, average degree is d=2m/n.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least d′=d/2
- Proof: Remove all nodes with degree < d′. Can only remove < nd′=nd/2=m edges so H′ non-empty.
- Consider node in H′:



- If length of all cycles is ≥5, the node has at least d′(d′-1) < n distinct neighbors of neighbors.

Spanners: Analysis

- If H has m edges, average degree is d=2m/n.
- Claim: H contains a non-empty subgraph H′ with minimum degree at least d′=d/2
- Proof: Remove all nodes with degree < d′. Can only remove < nd′=nd/2=m edges so H′ non-empty.
- Consider node in H′:



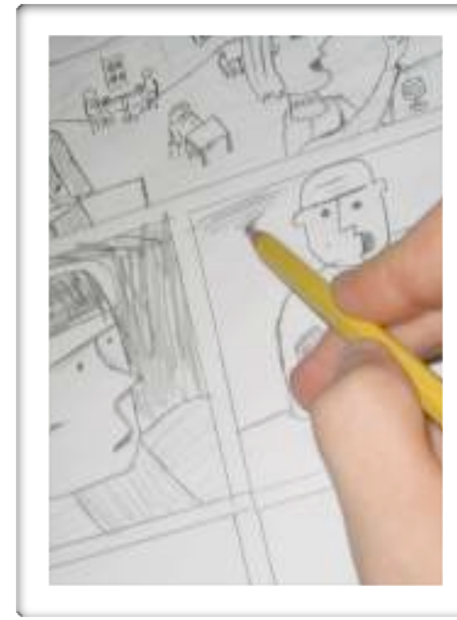
- If length of all cycles is ≥5, the node has at least d′(d′-1) < n distinct neighbors of neighbors.

Spanners Summary

- *Thm:* There's a $O(n^{1+1/t})$-space stream algorithm returns a (2t-1)-spanner. [Feigenbaum, Kannan, McGregor, Suri, Zhang 05]
- *Extension:* Can process weighted graphs by rounding weights and constructing spanners for each weight class.

I. **Spanners**

II. **Sparsifiers**



III. **Sketches**

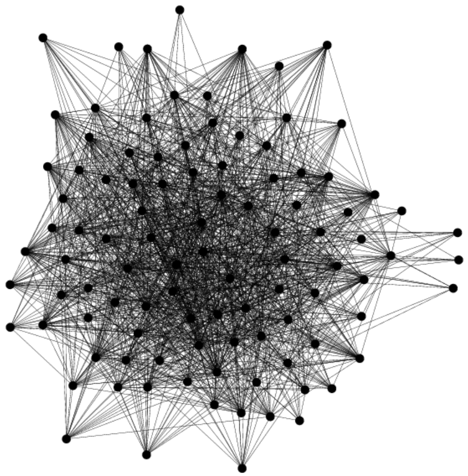
II. **Sparsifiers**

Synopsis for Cut Estimation
Merge-Reduce Stream Algorithm
Extensions

Sparsifiers & Cuts

Sparsifiers & Cuts

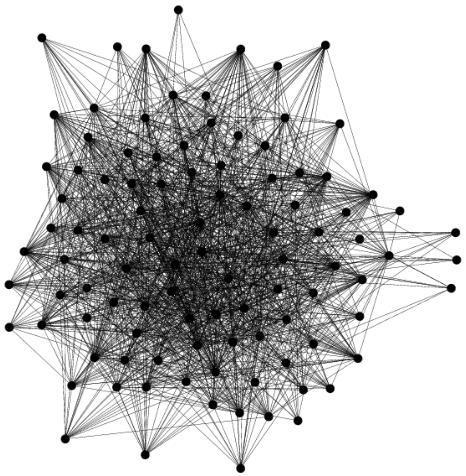
- *Measure:* Given a cut (L,R), the size of a cut $c_G(L,R)$ is the weight of all edges crossing the cut.



Original Graph G

Sparsifiers & Cuts

- *Measure:* Given a cut (L,R), the size of a cut $c_G(L,R)$ is the weight of all edges crossing the cut.



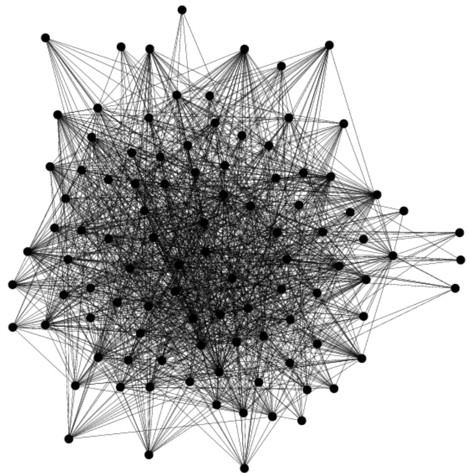
Original Graph G

- *Synopsis:* A subgraph H of G is a *(1+ε) sparsifier* if

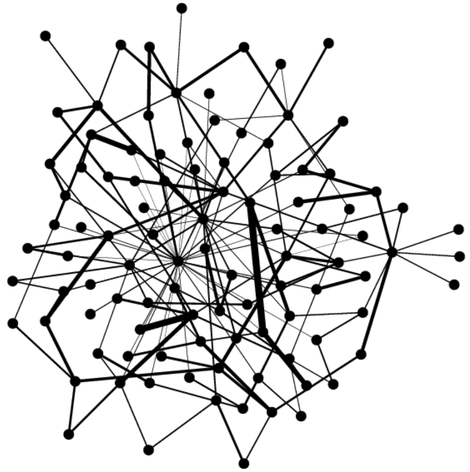
$$c_G(L,R) \leq c_H(L,R) \leq (1+\varepsilon)\, c_G(L,R) \quad \text{for all cuts.}$$

Sparsifiers & Cuts

- *Measure:* Given a cut (L,R), the size of a cut $c_G(L,R)$ is the weight of all edges crossing the cut.



Original Graph G



Sparsifier Graph H

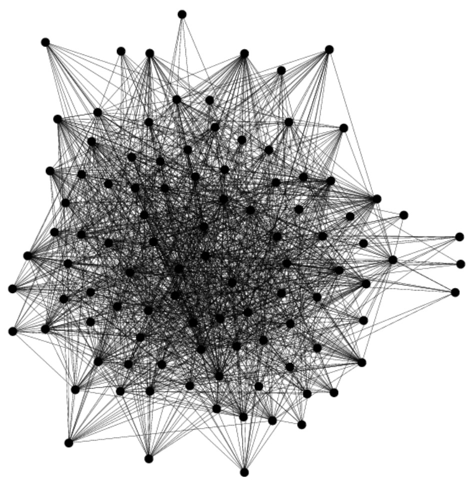
Graphs courtesy of Nick Harvey

- *Synopsis:* A subgraph H of G is a $(1+\varepsilon)$ *sparsifier* if

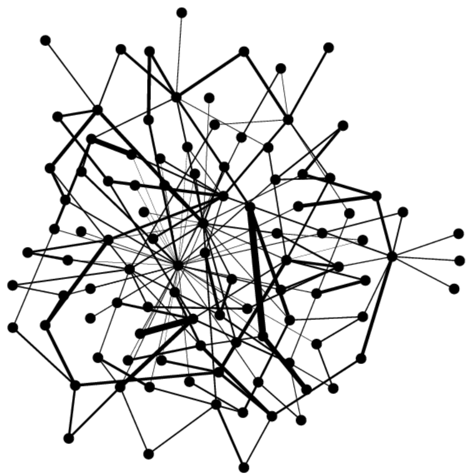
$$c_G(L,R) \leq c_H(L,R) \leq (1+\varepsilon)\, c_G(L,R) \quad \text{for all cuts.}$$

Sparsifiers & Cuts

- *Measure:* Given a cut (L,R), the size of a cut $c_G(L,R)$ is the weight of all edges crossing the cut.



Original Graph G



Sparsifier Graph H

Graphs courtesy of Nick Harvey

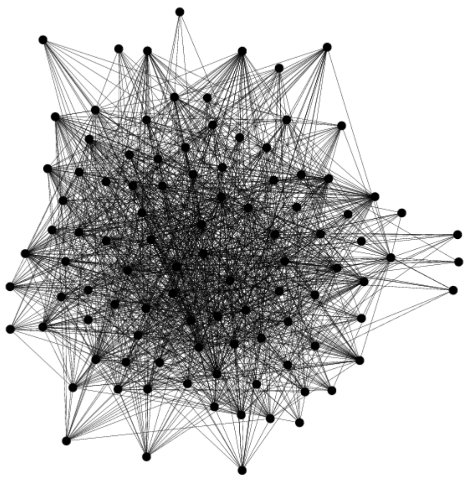
- *Synopsis:* A subgraph H of G is a (1+ε) *sparsifier* if

$$c_G(L,R) \leq c_H(L,R) \leq (1+\varepsilon)\, c_G(L,R) \quad \text{for all cuts.}$$

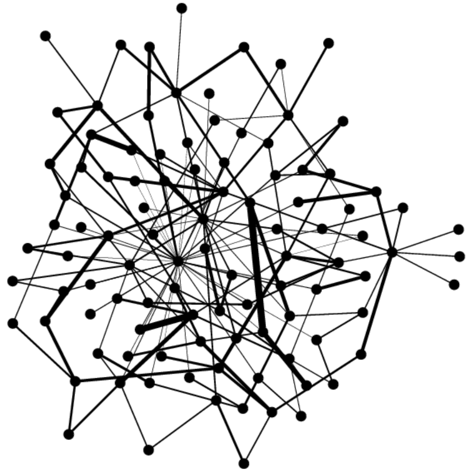
- *Thm (Benzur-Karger):* For any graph G there exists a (1+ε) sparsifier with only $O(\varepsilon^{-2} n)$ edges.

Sparsifiers & Cuts

- *Measure:* Given a cut (L,R), the size of a cut $c_G(L,R)$ is the weight of all edges crossing the cut.



Original Graph G



Sparsifier Graph H

Graphs courtesy of Nick Harvey

- *Synopsis:* A subgraph H of G is a $(1+\varepsilon)$ *sparsifier* if

$$c_G(L,R) \leq c_H(L,R) \leq (1+\varepsilon)\, c_G(L,R) \quad \text{for all cuts.}$$

- *Thm (Benzur-Karger):* For any graph G there exists a $(1+\varepsilon)$ sparsifier with only $O(\varepsilon^{-2} n)$ edges.
- *Thm:* Streaming construction in $O(\varepsilon^{-2} n \log^3 n)$ space.

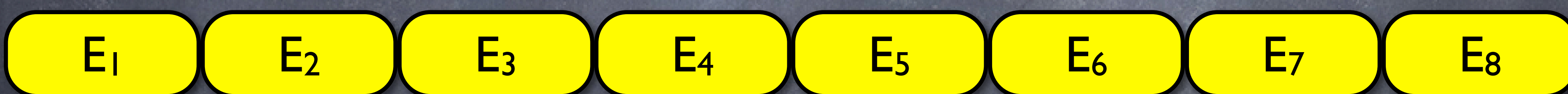
Sparsifier: Algorithm

Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, … each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.

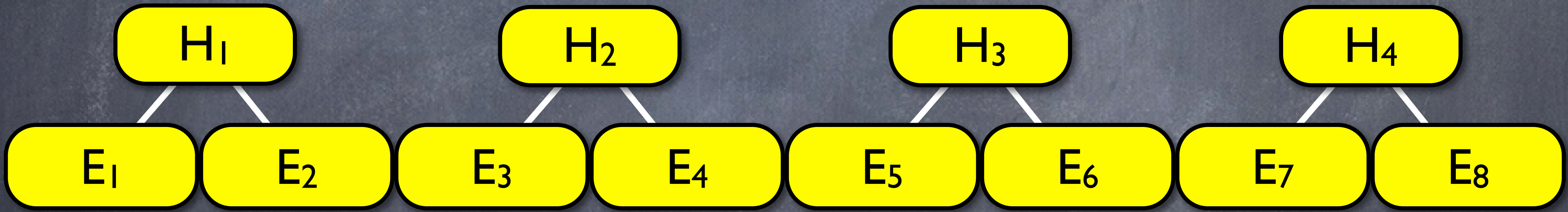
Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



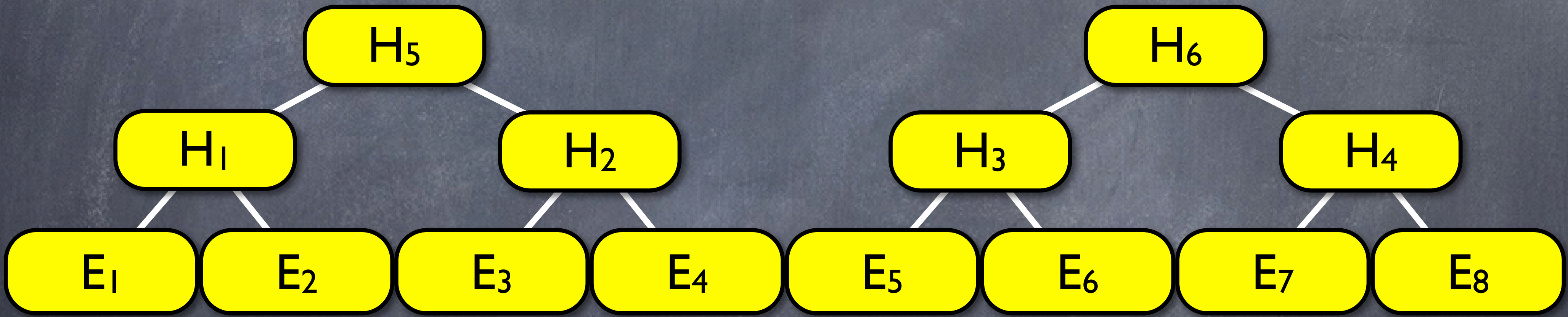
Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\epsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



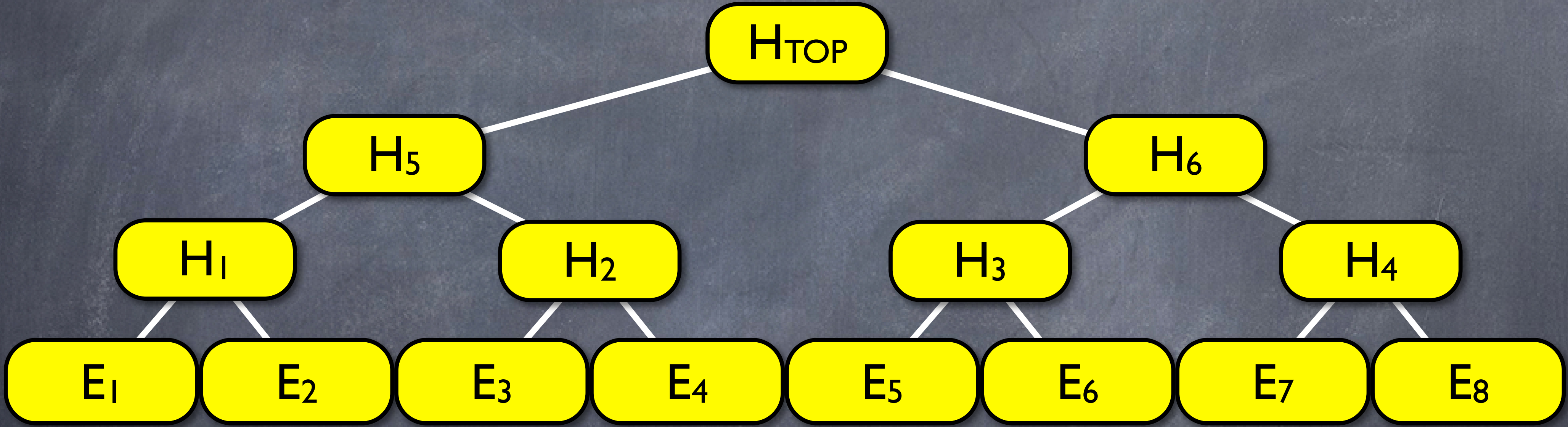
Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



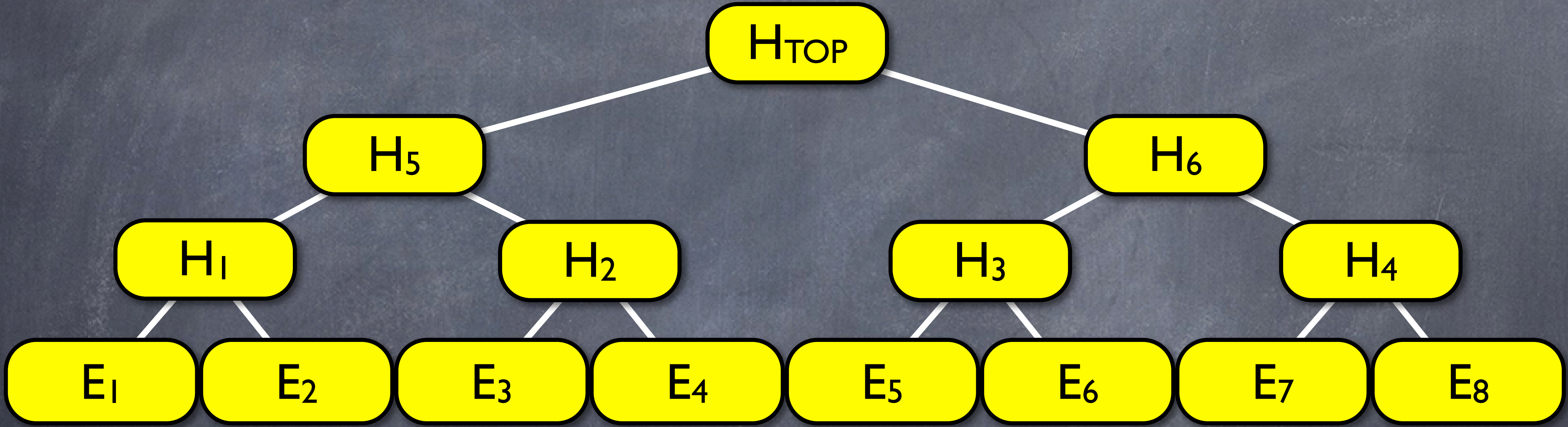
Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



Sparsifier: Algorithm

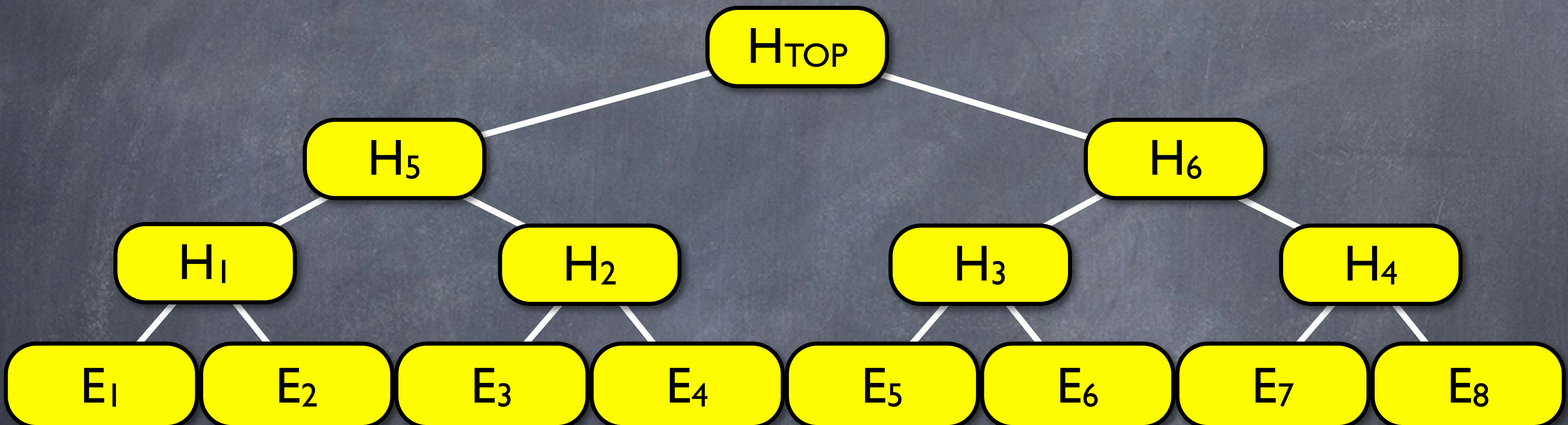
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.

Sparsifier: Algorithm

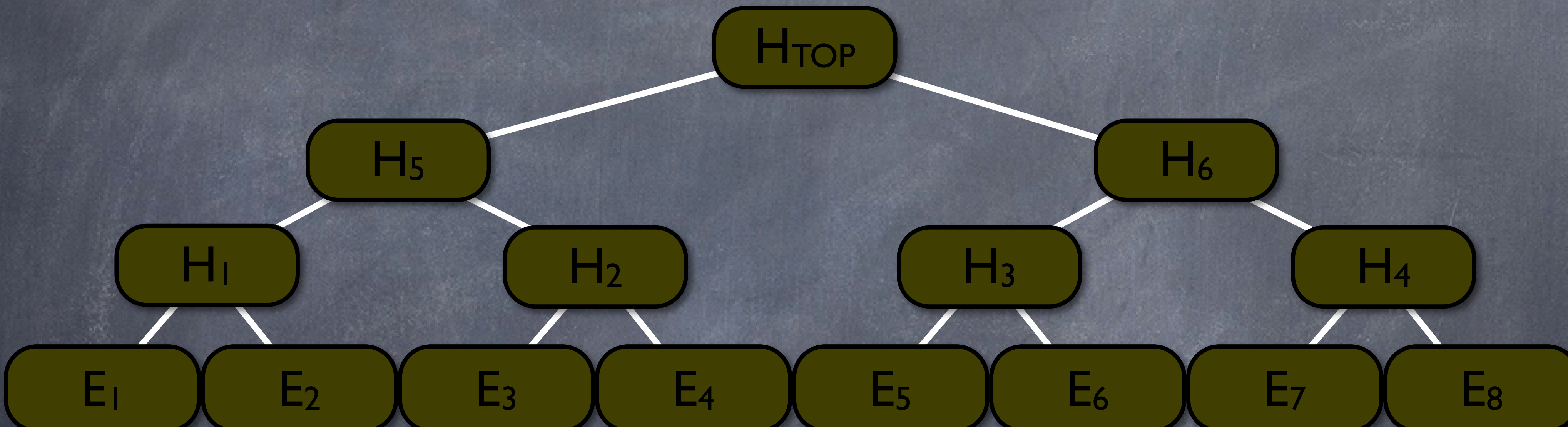
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

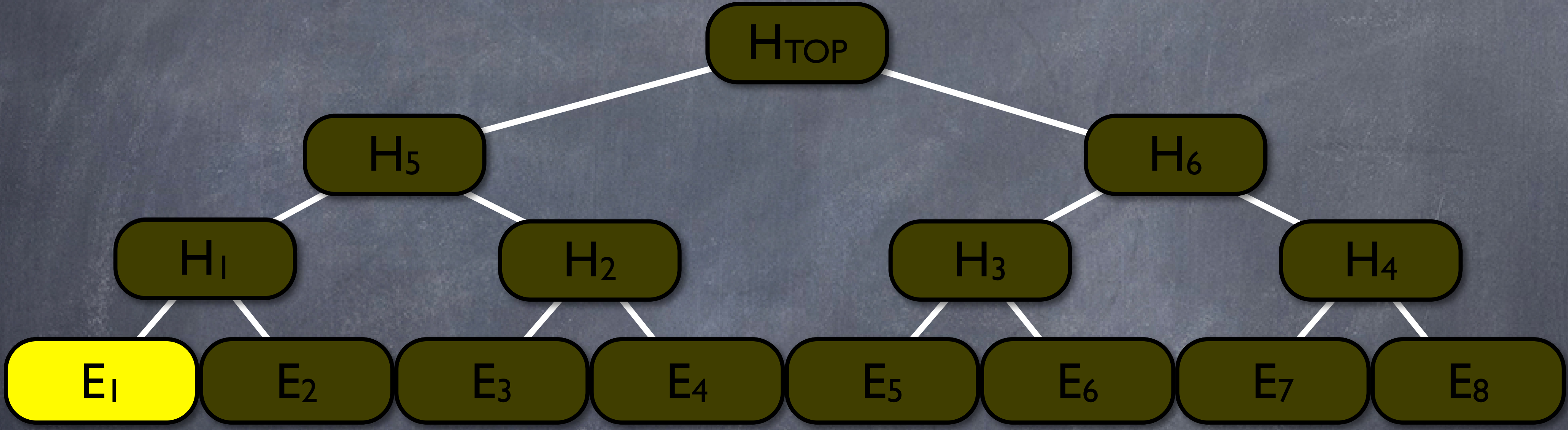
- **Main Idea:** Segment stream as $E_1, E_2, \ldots$ each of size $O(\varepsilon^{-2} n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

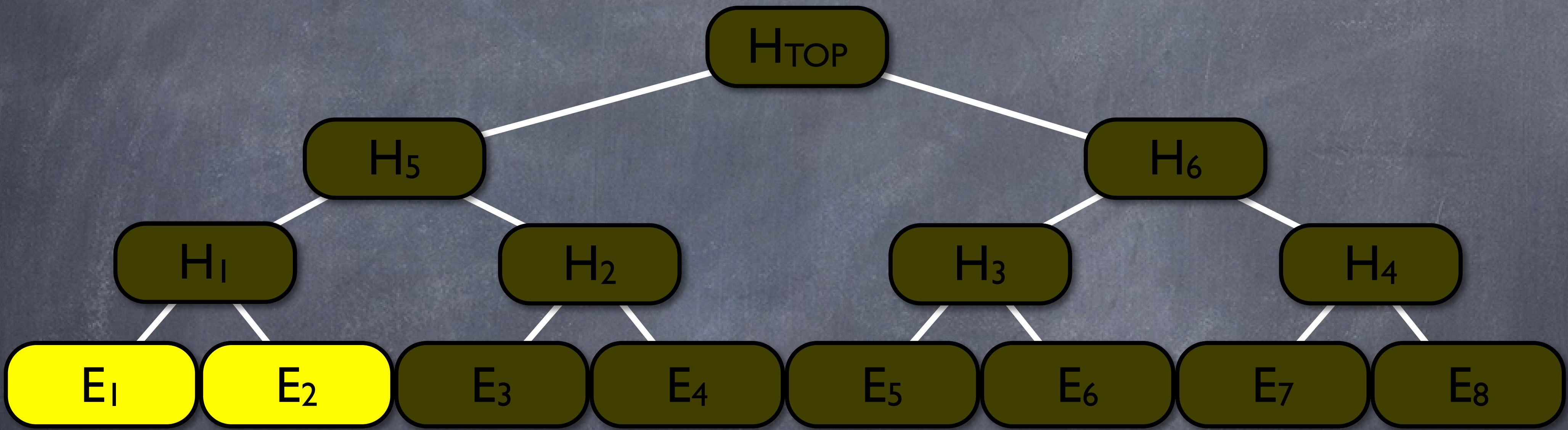
- **Main Idea:** Segment stream as $E_1, E_2, \ldots$ each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2} n \log n)$ memory.

Sparsifier: Algorithm

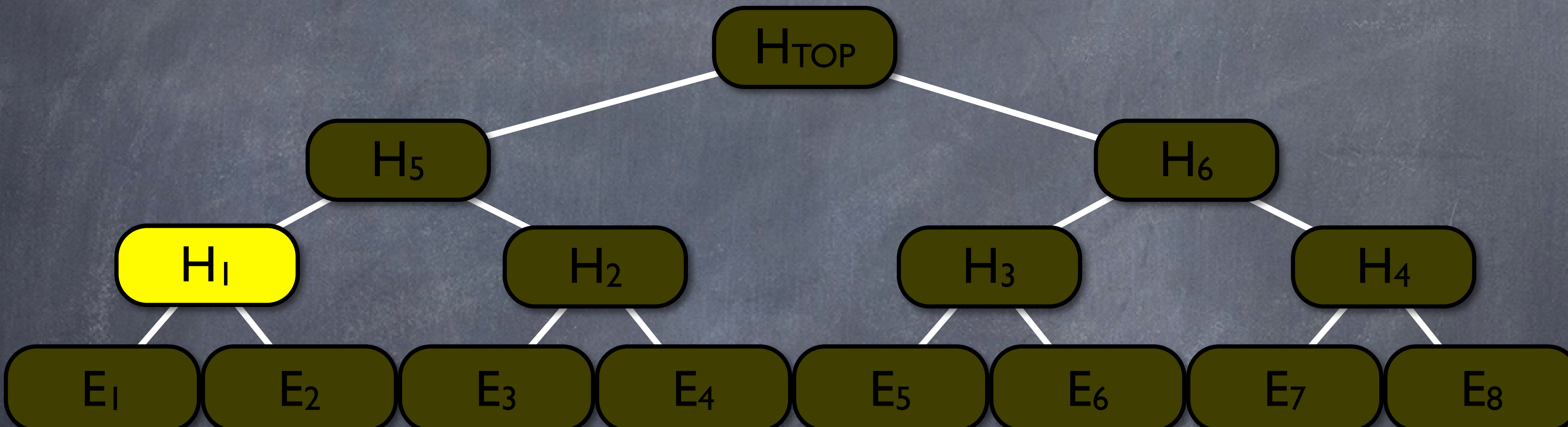
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

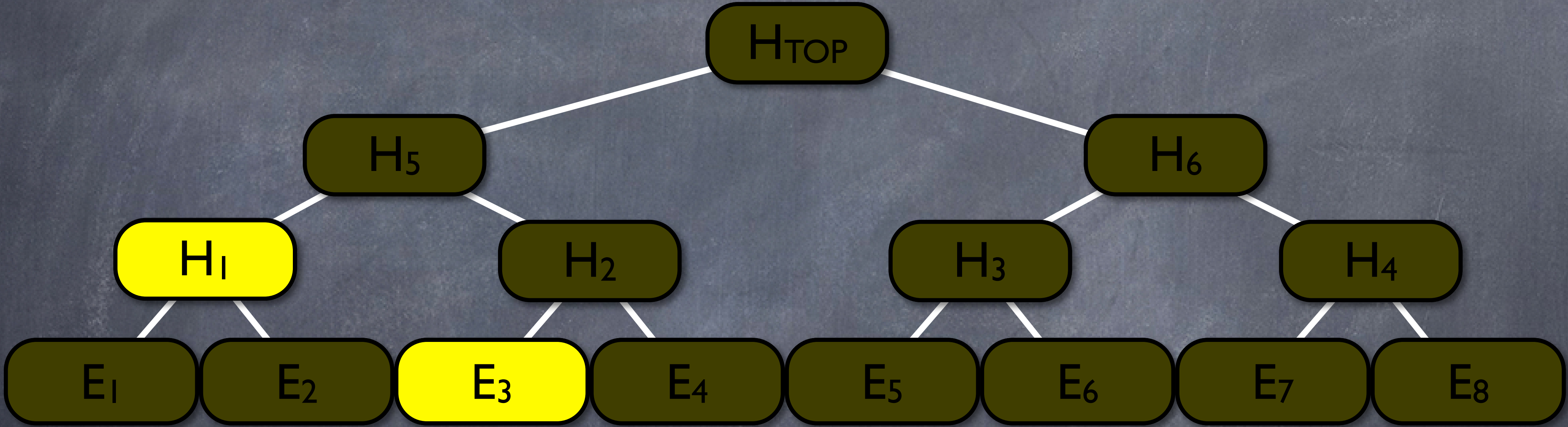
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

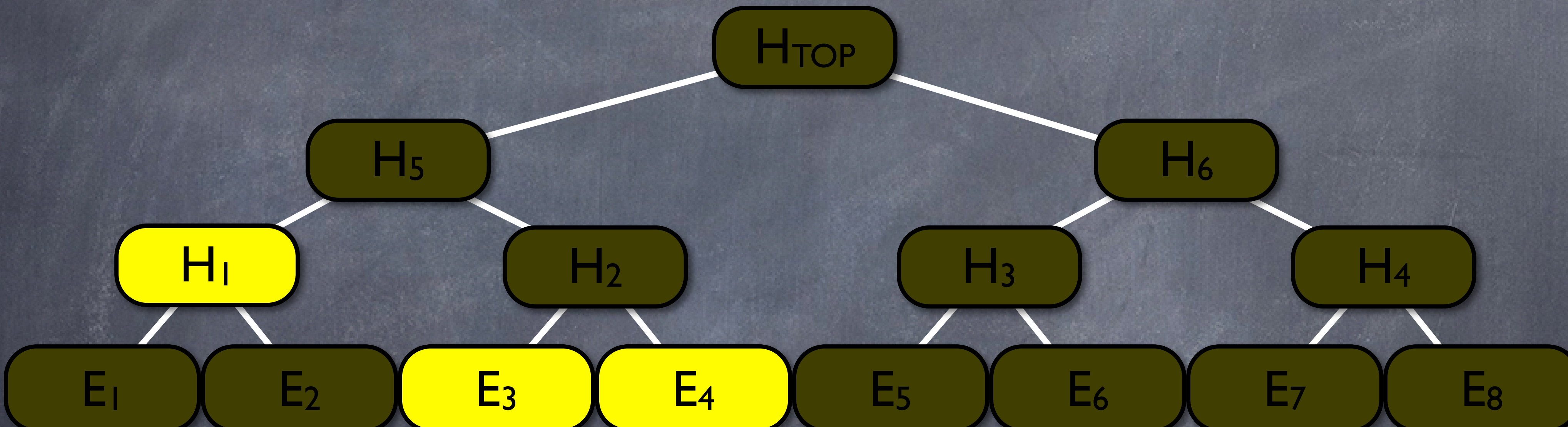
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2} n \log n)$ memory.

Sparsifier: Algorithm

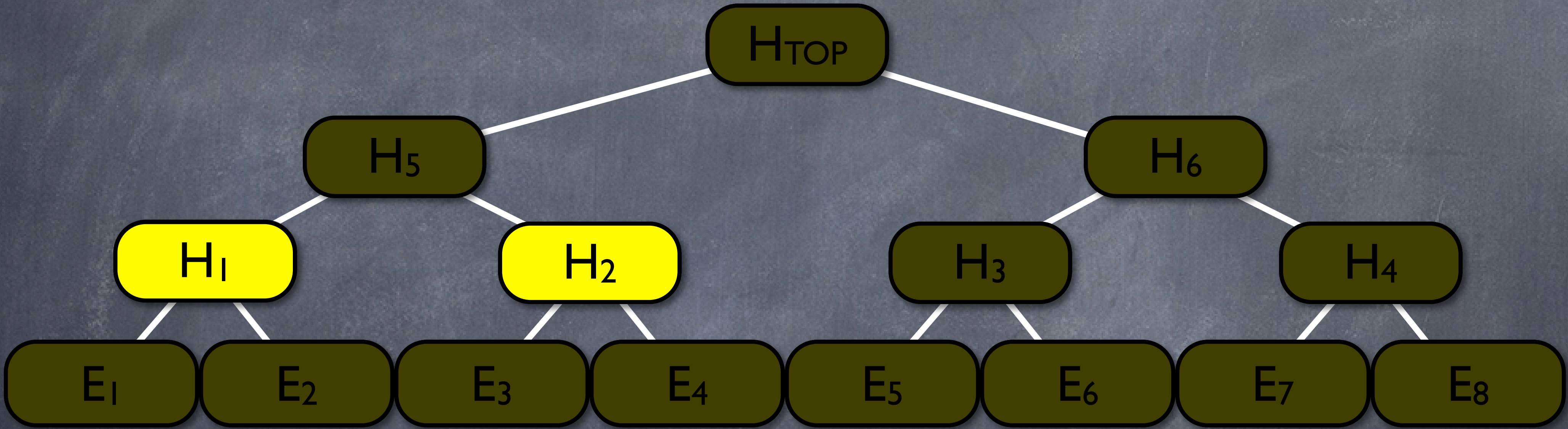
- **Main Idea:** Segment stream as $E_1, E_2, \ldots$ each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

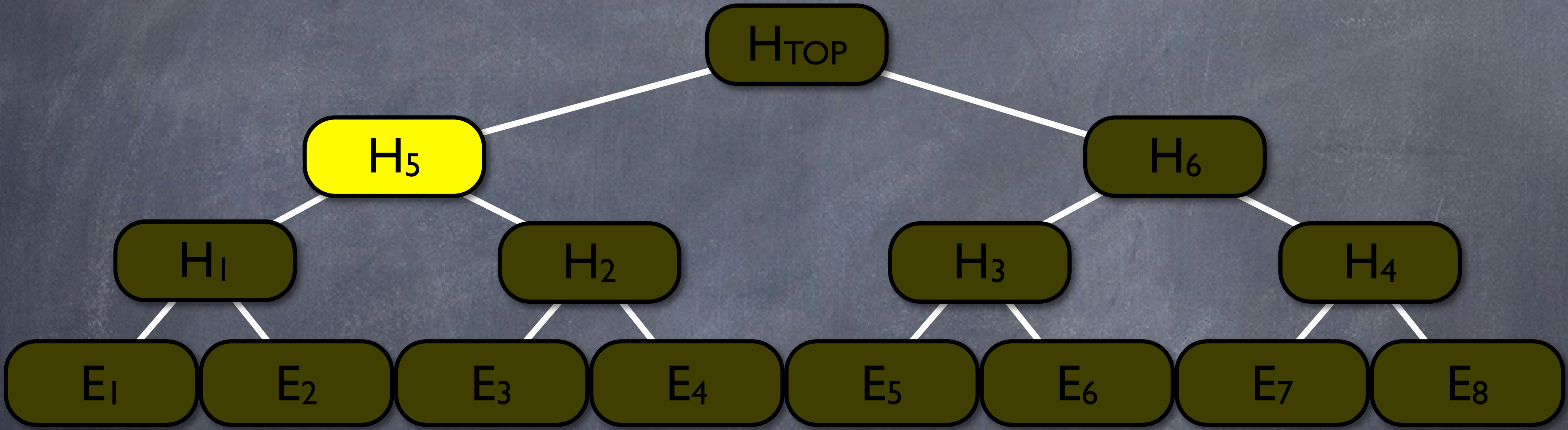
- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- Lemma: H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- Lemma: Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

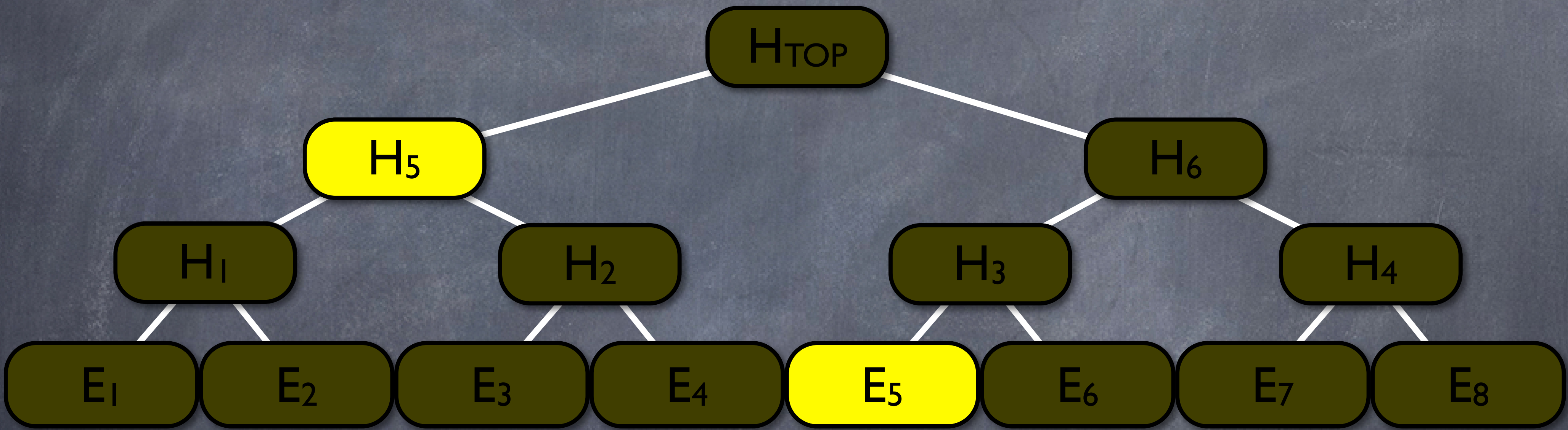
- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- Lemma: H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- Lemma: Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

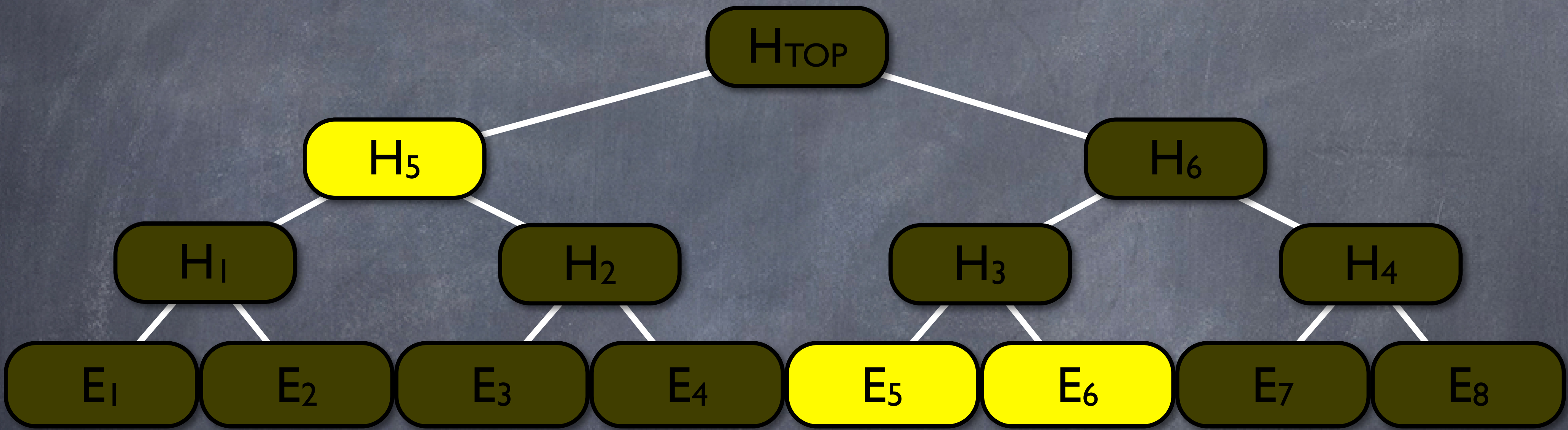
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

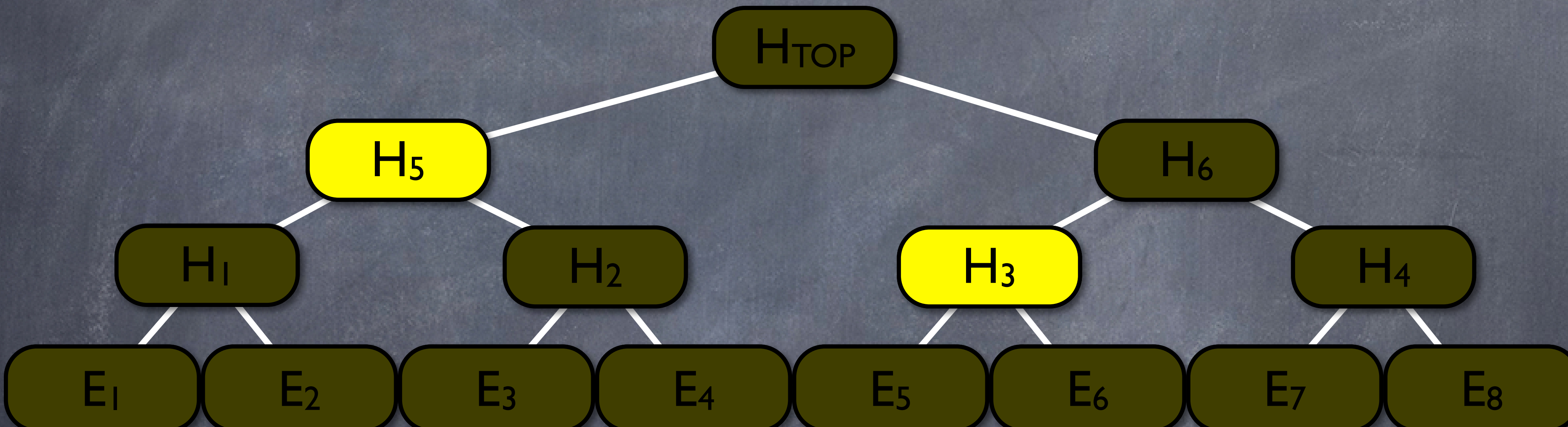
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\epsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\epsilon/\log n)$ yields a $(1+\epsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

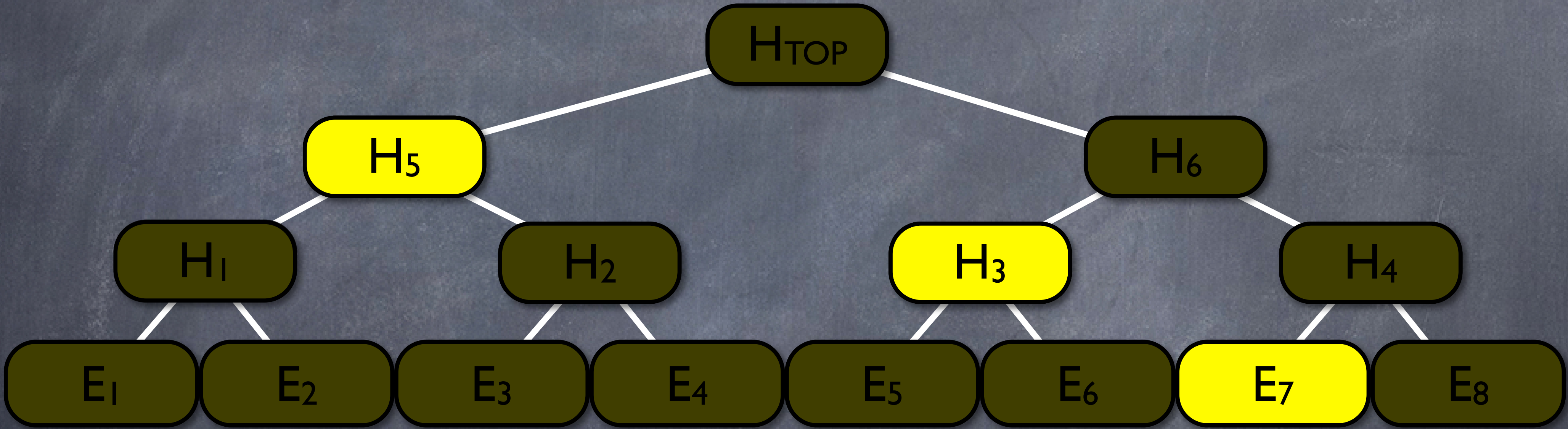
- **Main Idea:** Segment stream as $E_1, E_2, \ldots$ each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

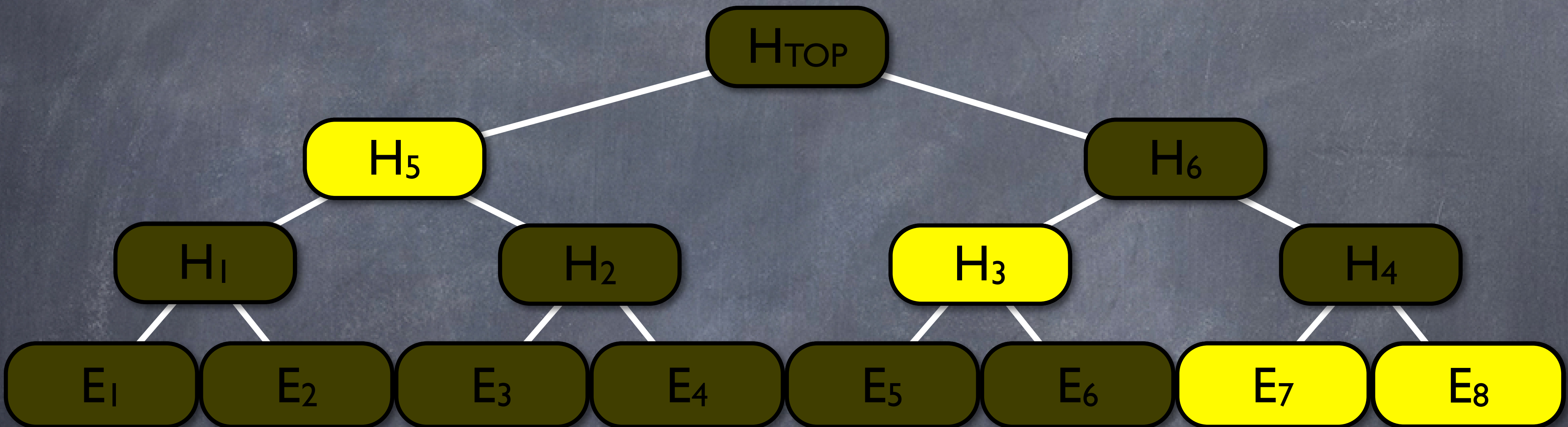
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

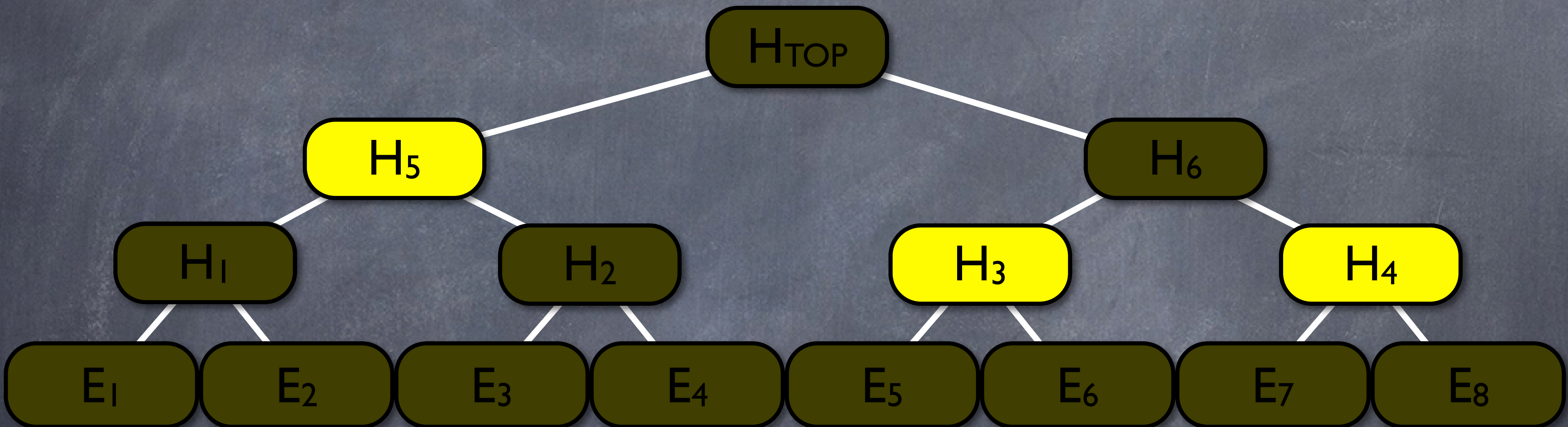
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2} n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

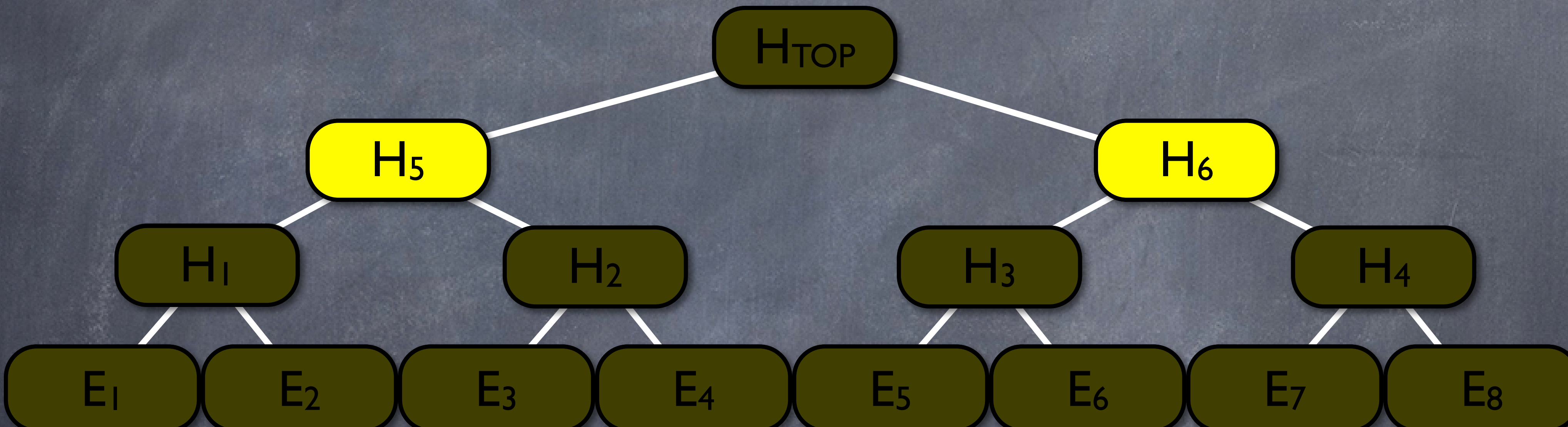
- **Main Idea:** Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

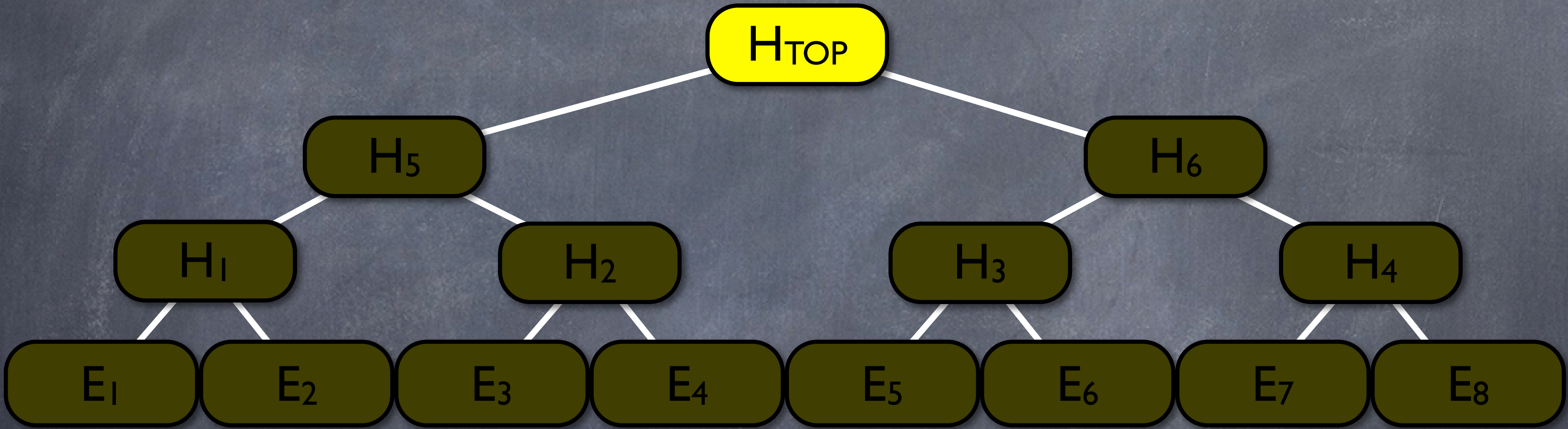
- **Main Idea:** Segment stream as E_1, E_2, … each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- **Lemma:** H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- **Lemma:** Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier: Algorithm

- Main Idea: Segment stream as E_1, E_2, ... each of size $O(\varepsilon^{-2}n)$. Let H_1 be $(1+\gamma)$ sparsifier of $E_1 \cup E_2$ etc.



- Lemma: H_{TOP} is a $(1+\gamma)^d$ sparsifier for $d=O(\log n)$. Setting $\gamma = O(\varepsilon/\log n)$ yields a $(1+\varepsilon)$ sparsifier.
- Lemma: Can find H_{TOP} with $O(\gamma^{-2}\, n \log n)$ memory.

Sparsifier Summary

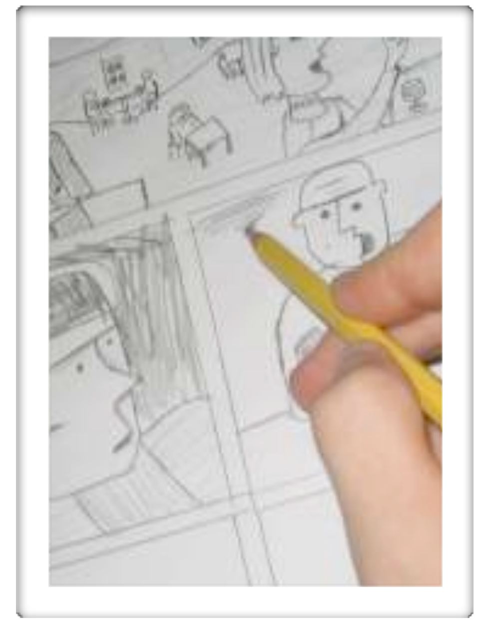
- *Thm:* A $(1+\varepsilon)$ sparsifier of a graph can be constructed in $O(\varepsilon^{-2}\, n \text{ polylog } n)$ space.

  [Ahn, Guha 09], [Goel, Kapralov, Khanna 10], [Sidiropoulos 10]

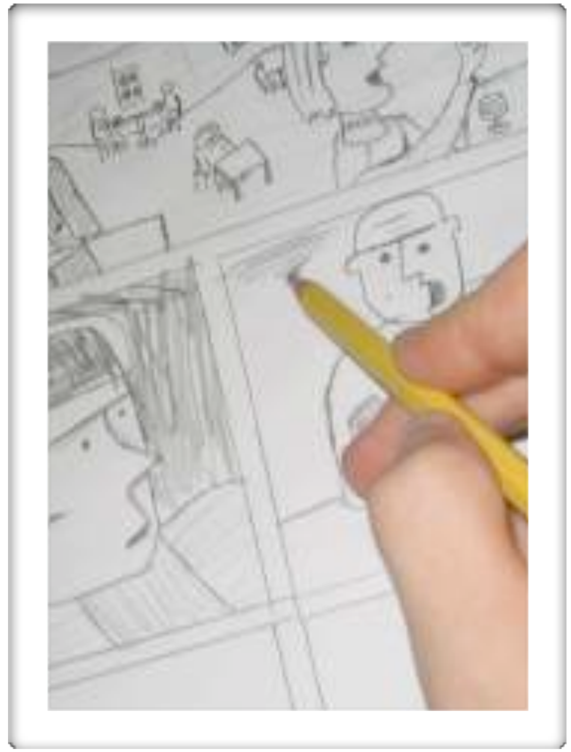
- Generalizes to spectral sparsification which preserves properties relating to random walks. [Kelner, Levin 11]

I. **Spanners**

II. **Sparsifiers**



III. **Sketches**



III. **Sketches**

Family of Linear Synopses
Distributed & Supports Deletions
Two Connectivity Examples

Linear Sketches

Linear Sketches

$$\begin{bmatrix} \\ \\ v \\ \\ \\ \end{bmatrix}$$

Linear Sketches

- <u>*Random linear projection:*</u> M: $\mathbb{R}^n \rightarrow \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\begin{bmatrix} \\ \\ v \\ \\ \\ \end{bmatrix}$$

Linear Sketches

- *Random linear projection:* M: $\mathbb{R}^n \rightarrow \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\begin{bmatrix} & & M & & \end{bmatrix} \begin{bmatrix} \\ v \\ \\ \\ \end{bmatrix}$$

Linear Sketches

- *Random linear projection:* M: $\mathbb{R}^n \rightarrow \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\begin{bmatrix} & & \\ & M & \\ & & \end{bmatrix} \begin{bmatrix} \\ \\ v \\ \\ \\ \end{bmatrix} = \begin{bmatrix} \\ Mv \\ \\ \end{bmatrix}$$

Linear Sketches

- <u>*Random linear projection:*</u> M: $\mathbb{R}^n \rightarrow \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\left[\quad M \quad\right]\left[\begin{matrix} \\ v \\ \\ \\ \end{matrix}\right] = \left[Mv \right] \longrightarrow \text{answer}$$

Linear Sketches

- *Random linear projection:* M: $\mathbb{R}^n \rightarrow \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\begin{bmatrix} & & \\ & M & \\ & & \end{bmatrix} \begin{bmatrix} \\ \\ v \\ \\ \\ \end{bmatrix} = \begin{bmatrix} Mv \end{bmatrix} \longrightarrow \text{answer}$$

- *Many results* for numerical statistics and basic geometric properties... *extensive theory* with connections to hashing, compressed sensing, dimensionality reduction, metric embeddings... *widely applicable* since embarrassingly parallelizable and suitable for stream processing.

Linear Sketches

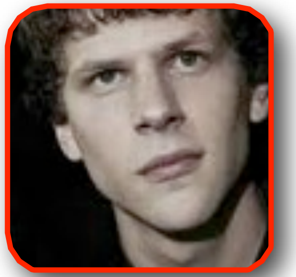
- *Random linear projection:* M: $\mathbb{R}^n \to \mathbb{R}^k$ (where $k \ll n$) that preserves properties of any $v \in \mathbb{R}^n$ with high probability.

$$\begin{bmatrix} & & \\ & M & \\ & & \end{bmatrix} \begin{bmatrix} \\ \\ v \\ \\ \\ \end{bmatrix} = \begin{bmatrix} Mv \end{bmatrix} \longrightarrow \text{answer}$$

- *Many results* for numerical statistics and basic geometric properties... *extensive theory* with connections to hashing, compressed sensing, dimensionality reduction, metric embeddings... *widely applicable* since embarrassingly parallelizable and suitable for stream processing.

? *Question:* What about analyzing massive graphs via sketches?

Distributed Data

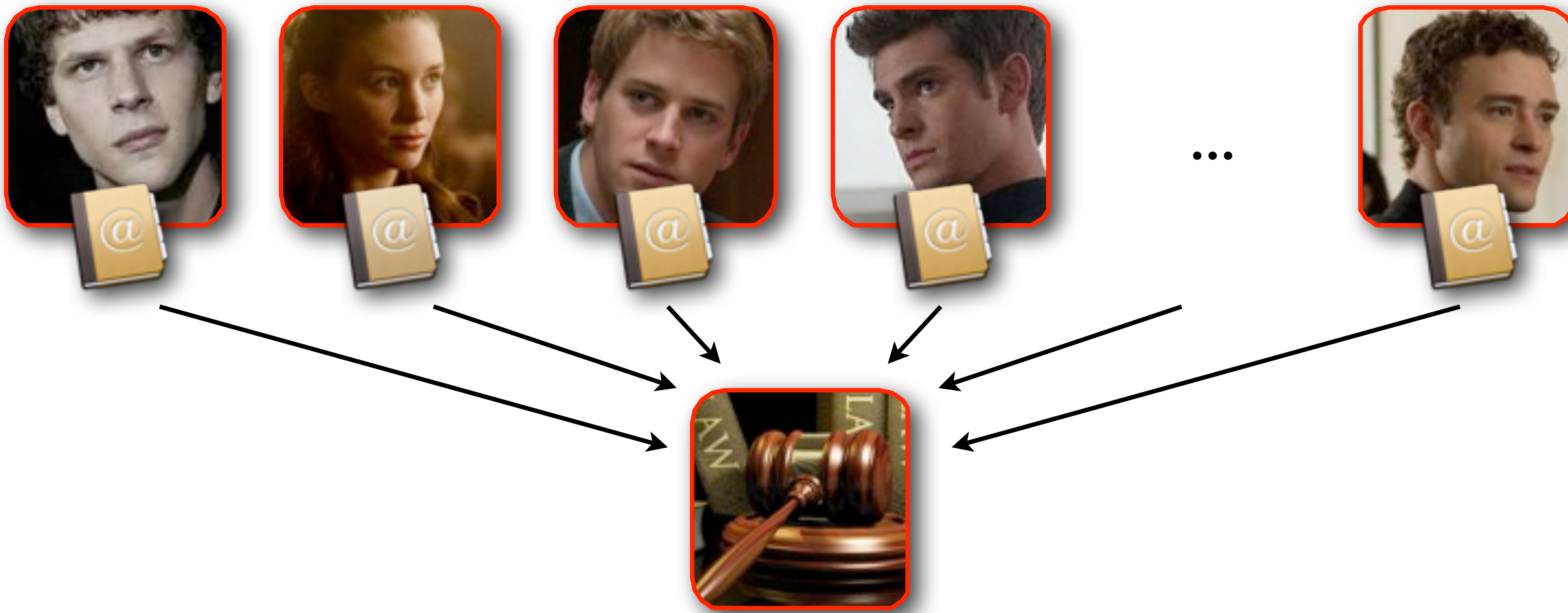



...

Distributed Data

- *Input:* Each player knows neighborhood $\Gamma(v)$ for a node v

Distributed Data



- ***Input:*** Each player knows neighborhood Γ(v) for a node v
- ***Goal:*** Simultaneously, each player sends O(polylog n) bits to a central player who then determines if graph is connected.

This can't be possible?!

- Suppose there's a *bridge* (u,v) in the graph, i.e., a friendship that is essential to ensuring the graph is connected.

This can't be possible?!

- Suppose there's a *bridge* (u,v) in the graph, i.e., a friendship that is essential to ensuring the graph is connected.
- *Dubious Claim:* At least one player needs to send $\Omega(n)$ bits.

This can't be possible?!

- Suppose there's a *bridge* (u,v) in the graph, i.e., a friendship that is essential to ensuring the graph is connected.
- *Dubious Claim:* At least one player needs to send Ω(n) bits.

a) Central player needs to know about the special friendship.

This can't be possible?!

- Suppose there's a *bridge* (u,v) in the graph, i.e., a friendship that is essential to ensuring the graph is connected.
- *Dubious Claim:* At least one player needs to send Ω(n) bits.
  a) Central player needs to know about the special friendship.
  b) Participant doesn't know which friendships are special.

This can't be possible?!

- Suppose there's a *bridge* (u,v) in the graph, i.e., a friendship that is essential to ensuring the graph is connected.
- *Dubious Claim:* At least one player needs to send Ω(n) bits.
  a) Central player needs to know about the special friendship.
  b) Participant doesn't know which friendships are special.
  c) Participants may have Ω(n) friends.

How to do it...

How to do it...

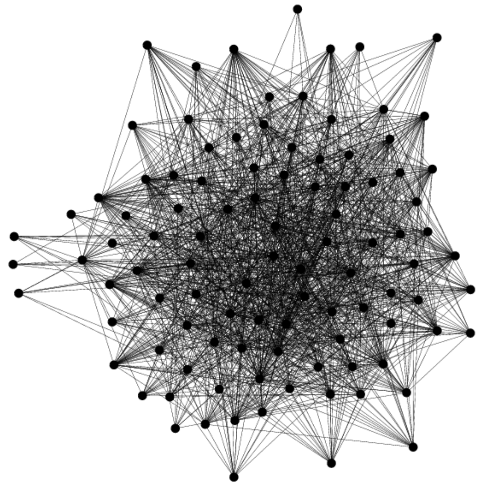
- Players send carefully-designed sketches of address books.

How to do it...

- Players send carefully-designed sketches of address books.
- *Main Idea:* Exploit homomorphic properties of linear sketches and emulate a classical algorithm in *sketch space*.

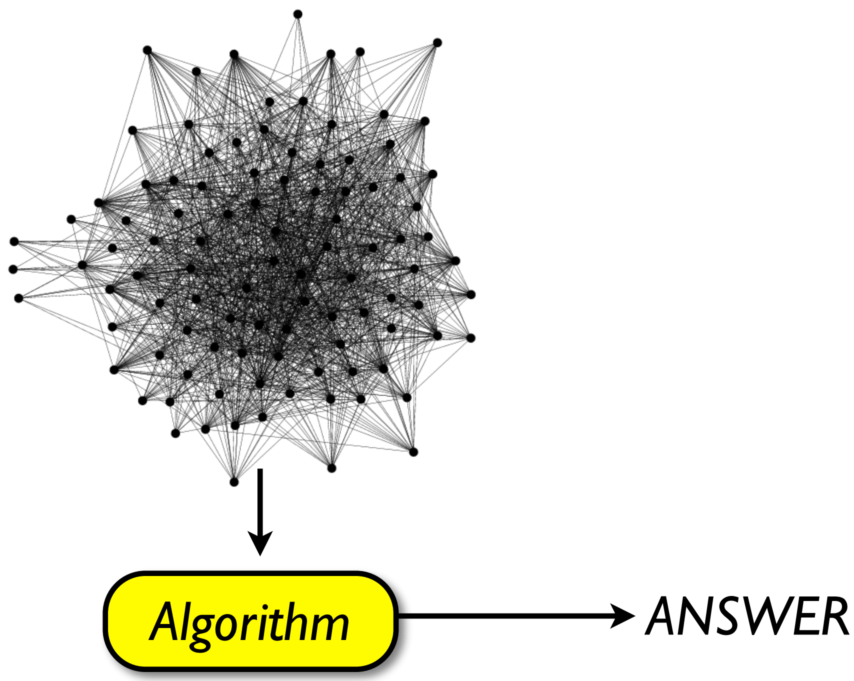
How to do it...

- Players send carefully-designed sketches of address books.
- *Main Idea:* Exploit homomorphic properties of linear sketches and emulate a classical algorithm in *sketch space*.



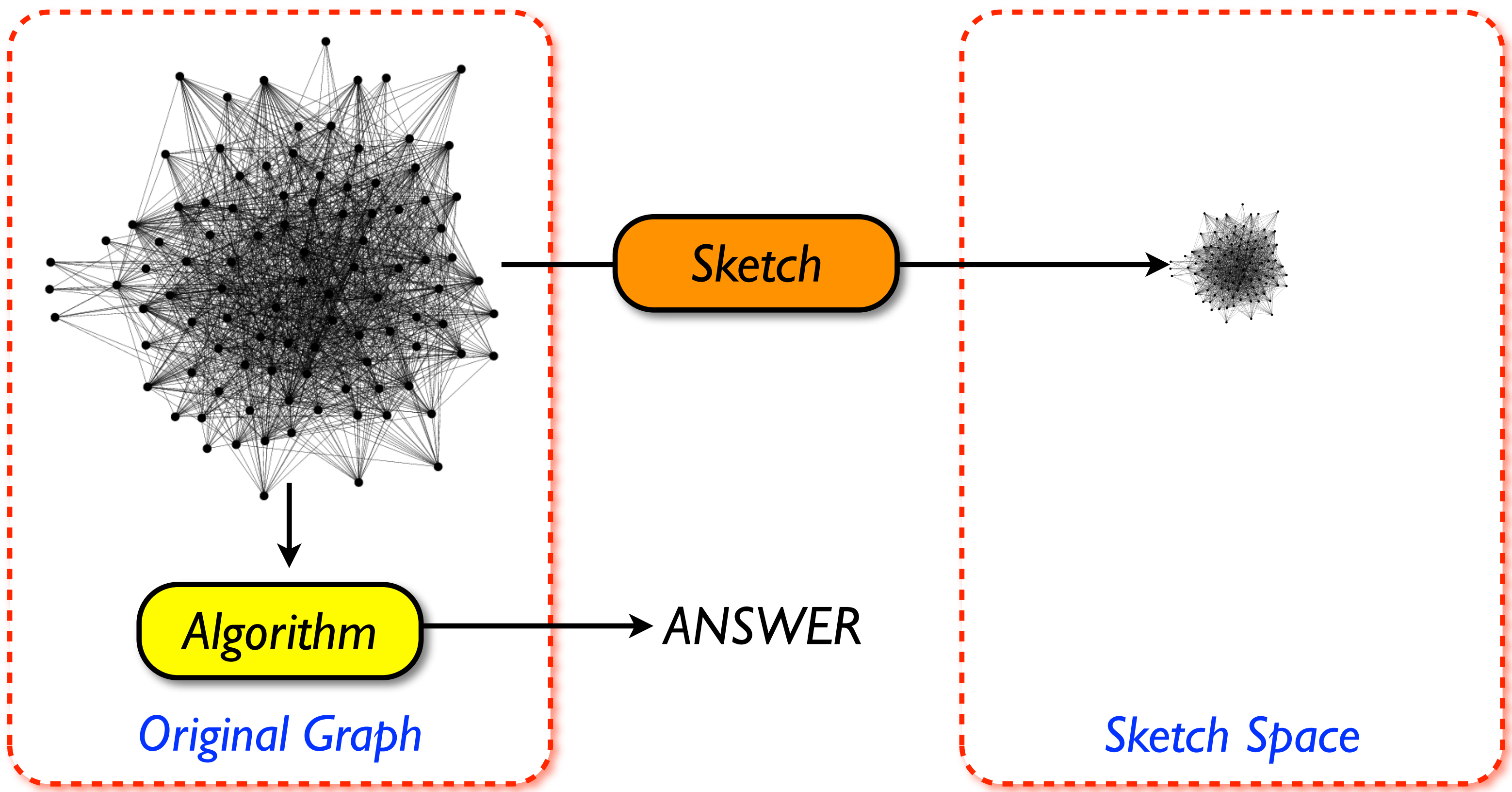
How to do it...

- Players send carefully-designed sketches of address books.
- *Main Idea:* Exploit homomorphic properties of linear sketches and emulate a classical algorithm in *sketch space*.



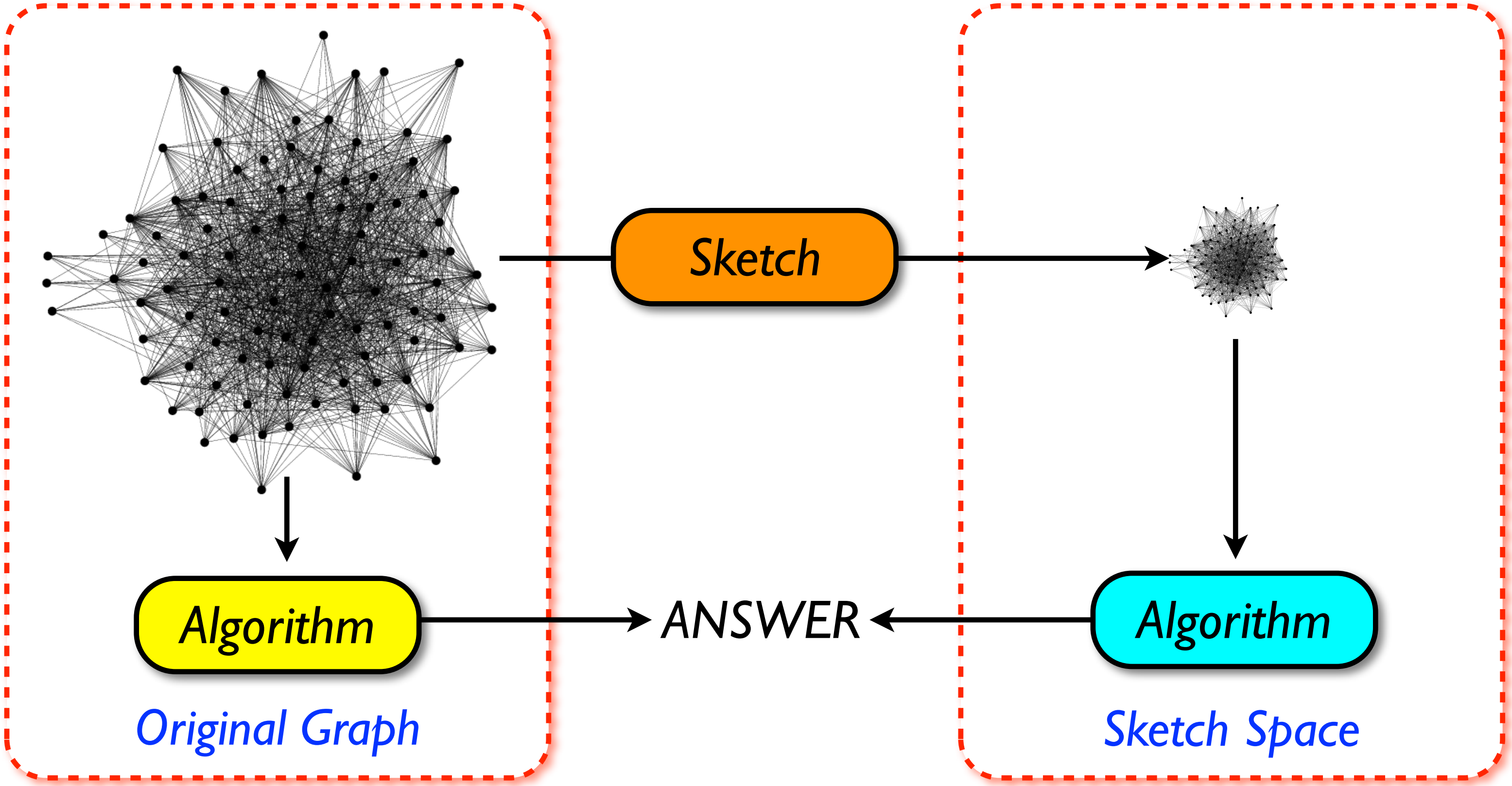
How to do it...

- Players send carefully-designed sketches of address books.
- *Main Idea:* Exploit homomorphic properties of linear sketches and emulate a classical algorithm in *sketch space*.



How to do it...

- Players send carefully-designed sketches of address books.
- *Main Idea:* Exploit homomorphic properties of linear sketches and emulate a classical algorithm in *sketch space*.



Two Examples

First Theorem: Testing Connectivity

a) *Dynamic Graph Stream:* O(n polylog n) space.

b) *Distributed Setting:* O(polylog n) length messages.

Second Theorem: Checking every cut has size $\geq$ k

a) *Dynamic Graph Stream:* O(n k polylog n) space.

b) *Distributed Setting:* O(k polylog n) length.

Ingredient 1: Basic Algorithm

Ingredient 1: Basic Algorithm

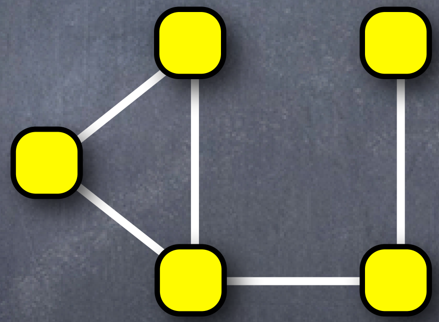
- Algorithm (Spanning Forest):

Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge

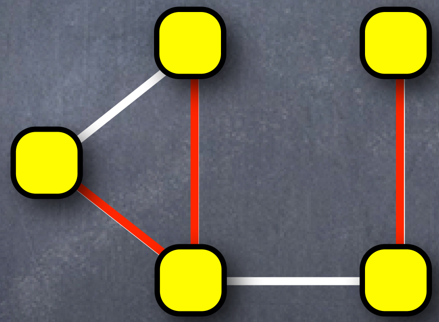
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge



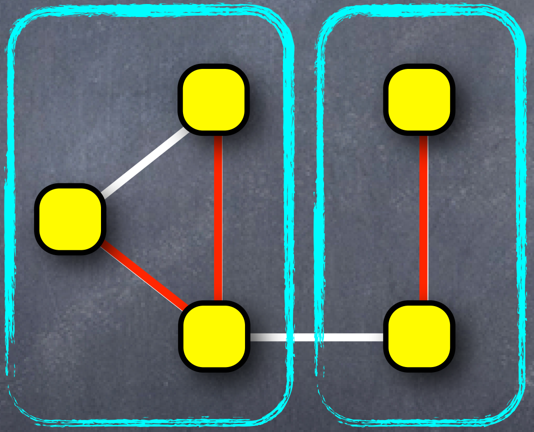
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge



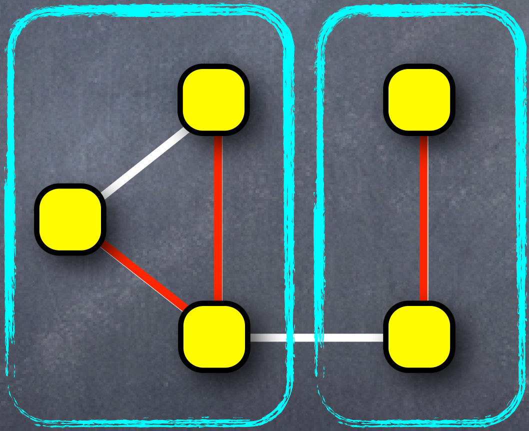
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge



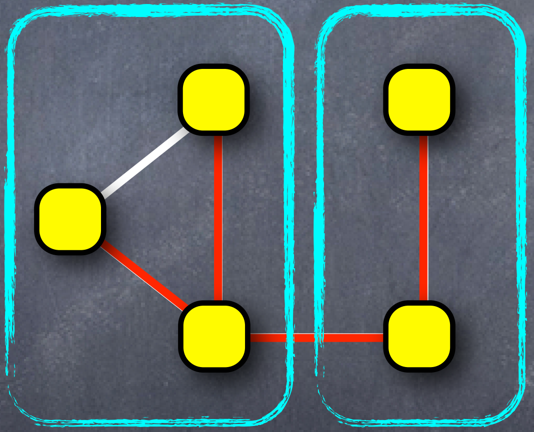
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge
  2. For each connected comp: pick incident edge



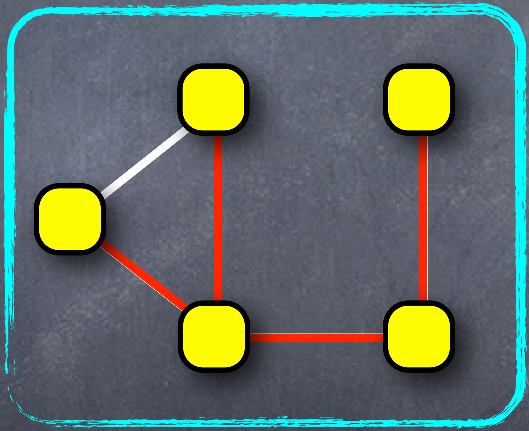
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge
  2. For each connected comp: pick incident edge



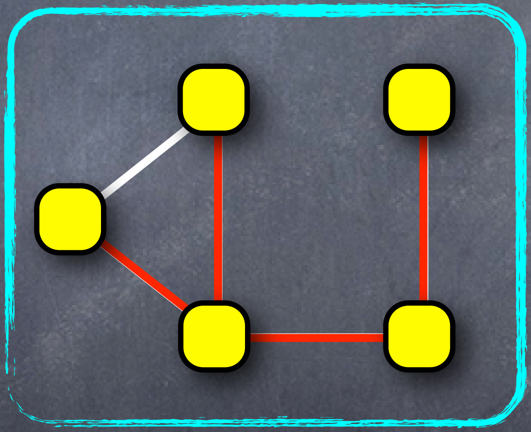
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge
  2. For each connected comp: pick incident edge



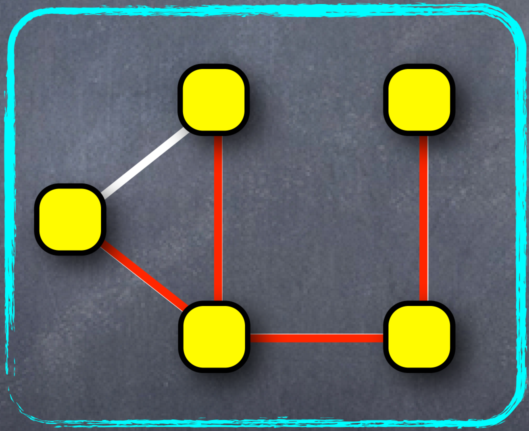
Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
  1. For each node: pick incident edge
  2. For each connected comp: pick incident edge
  3. Repeat until no edges between connected comp.



Ingredient 1: Basic Algorithm

- Algorithm (Spanning Forest):
    1. For each node: pick incident edge
    2. For each connected comp: pick incident edge
    3. Repeat until no edges between connected comp.



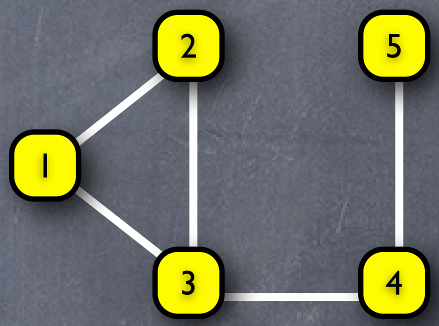
- Lemma: After O(log n) rounds selected edges include spanning forest.

Ingredient 2: Sketching Neighborhoods

Ingredient 2: Sketching Neighborhoods

- For node i, let a_i be vector indexed by node pairs. Non-zero entries: $a_i[i,j]=1$ if $j>i$ and $a_i[i,j]=-1$ if $j<i$.

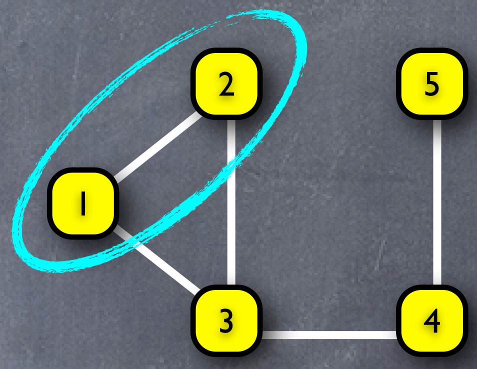
$$
\begin{array}{rcccccccccc}
 & \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\
\mathbf{a}_1 = & \Big(\; 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \; \Big) \\
\mathbf{a}_2 = & \Big(\; -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \; \Big)
\end{array}
$$



Ingredient 2: Sketching Neighborhoods

- For node i, let a_i be vector indexed by node pairs. Non-zero entries: $a_i[i,j]=1$ if $j>i$ and $a_i[i,j]=-1$ if $j<i$.

$$
\begin{array}{rcccccccccc}
 & \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\
\mathbf{a}_1 = (& 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \;) \\
\mathbf{a}_2 = (& -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \;)
\end{array}
$$

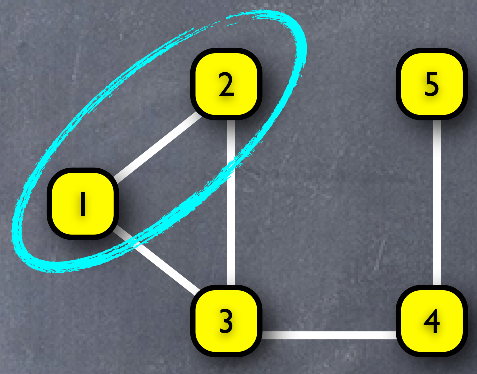


Ingredient 2: Sketching Neighborhoods

- For node i, let a_i be vector indexed by node pairs. Non-zero entries: $a_i[i,j]=1$ if $j>i$ and $a_i[i,j]=-1$ if $j<i$.

$$\begin{array}{rcccccccccc}
 & \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\}
\end{array}$$

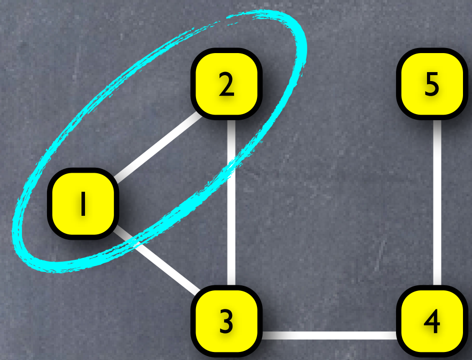
$$\mathbf{a}_1 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$
$$\mathbf{a}_2 = \begin{pmatrix} -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$
$$\mathbf{a}_1 + \mathbf{a}_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Ingredient 2: Sketching Neighborhoods

- For node i, let a_i be vector indexed by node pairs. Non-zero entries: $a_i[i,j]=1$ if $j>i$ and $a_i[i,j]=-1$ if $j<i$.

$$
\begin{array}{r@{\;}c@{\;}c}
 & & \begin{array}{cccccccccc} \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \end{array} \\
\mathbf{a}_1 & = & \left(\begin{array}{cccccccccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\
\mathbf{a}_2 & = & \left(\begin{array}{cccccccccc} -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\
\mathbf{a}_1 + \mathbf{a}_2 & = & \left(\begin{array}{cccccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right)
\end{array}
$$



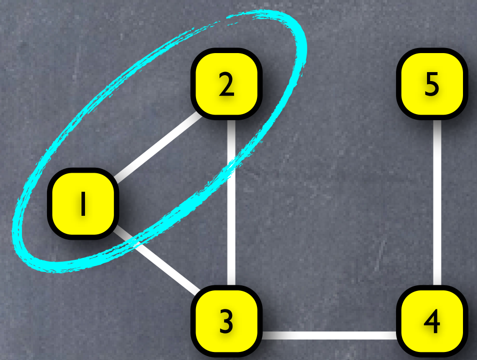
- Lemma: For any subset of nodes $S \subset V$,

$$\text{support}\left(\sum_{i \in S} \mathbf{a}_i\right) = E(S, V \setminus S)$$

Ingredient 2: Sketching Neighborhoods

- For node i, let a_i be vector indexed by node pairs. Non-zero entries: $a_i[i,j]=1$ if $j>i$ and $a_i[i,j]=-1$ if $j<i$.

$$
\begin{array}{r|cccccccccc|}
 & \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\
\mathbf{a}_1 = & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{a}_2 = & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\mathbf{a}_1 + \mathbf{a}_2 = & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}
$$



- **Lemma:** For any subset of nodes $S \subset V$,

$$\text{support}\left(\sum_{i \in S} \mathbf{a}_i\right) = E(S, V \setminus S)$$

- **Lemma:** $\exists$ random M: $\mathbb{R}^N \to \mathbb{R}^k$ with k=O(polylog N) such that for any $a \in \mathbb{R}^N$, with high probability

$$M\mathbf{a} \longrightarrow e \in \text{support}(\mathbf{a})$$

Recipe: Sketch & Compute on Sketches

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:
  - Use Ma_j to get incident edge on each node j

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:
  - Use Ma_j to get incident edge on each node j
  - For i=2 to log n:
    - To get incident edge on component $S \subset V$ use:

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:
  - Use Ma_j to get incident edge on each node j
  - For i=2 to log n:
    - To get incident edge on component $S \subset V$ use:

$$\sum_{j \in S} M\mathbf{a}_j = M\left(\sum_{j \in S} \mathbf{a}_j\right)$$

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:
  - Use Ma_j to get incident edge on each node j
  - For i=2 to log n:
    - To get incident edge on component $S \subset V$ use:

$$\sum_{j \in S} M\mathbf{a}_j = M(\sum_{j \in S} \mathbf{a}_j) \longrightarrow e \in \text{support}(\sum_{j \in S} \mathbf{a}_j) = E(S, V \setminus S)$$

Recipe: Sketch & Compute on Sketches

- Sketch: Each player sends Ma_j
- Central Player Runs Algorithm in Sketch Space:
  - Use Ma_j to get incident edge on each node j
  - For i=2 to log n:
    - To get incident edge on component $S \subset V$ use:

$$\sum_{j \in S} M\mathbf{a}_j = M(\sum_{j \in S} \mathbf{a}_j) \longrightarrow e \in \text{support}(\sum_{j \in S} \mathbf{a}_j) = E(S, V \setminus S)$$

Detail: Actually each player sends log n indept sketches M_1a_j, M_2a_j, … and central player uses M_ia_j when emulating i^{th} iteration of the algorithm.

Two Examples

First Theorem: Testing Connectivity

a) *Dynamic Graph Stream:* O(n polylog n) space.

b) *Distributed Setting:* O(polylog n) length messages.

Second Theorem: Checking every cut has size $\geq$ k

a) *Dynamic Graph Stream:* O(n k polylog n) space.

b) *Distributed Setting:* O(k polylog n) length.

Two Examples

First Theorem: Testing Connectivity

a) *Dynamic Graph Stream:* O(n polylog n) space.

b) *Distributed Setting:* O(polylog n) length messages.

Second Theorem: Checking every cut has size $\geq$ k

a) *Dynamic Graph Stream:* O(n k polylog n) space.

b) *Distributed Setting:* O(k polylog n) length.

Ingredient 1: Basic Algorithm

Ingredient 1: Basic Algorithm

- Algorithm (k-Connectivity):

Ingredient 1: Basic Algorithm

- Algorithm (k-Connectivity):

  1. Let F_1 be spanning forest of $G(V,E)$

Ingredient 1: Basic Algorithm

- Algorithm (k-Connectivity):
  1. Let F_1 be spanning forest of $G(V,E)$
  2. For $i=2$ to k:
     2.1. Let F_i be spanning forest of $G(V,E-F_1-\ldots-F_{i-1})$

Ingredient 1: Basic Algorithm

- Algorithm (k-Connectivity):
  1. Let F_1 be spanning forest of $G(V,E)$
  2. For $i=2$ to k:
     2.1. Let F_i be spanning forest of $G(V,E-F_1-\ldots-F_{i-1})$
- Lemma: $G(V,F_1+\ldots+F_k)$ is k-connected iff $G(V,E)$ is.

Ingredient 2: Connectivity Sketches

Ingredient 2: Connectivity Sketches

- Sketch: Simultaneously construct k independent connectivity sketches $\{M_1G, M_2G, \ldots M_kG\}$.

Ingredient 2: Connectivity Sketches

- Sketch: Simultaneously construct k independent connectivity sketches $\{M_1G, M_2G, \ldots M_kG\}$.
- Run Algorithm in Sketch Space:
  - Use M_1G to find a spanning forest F_1 of G

Ingredient 2: Connectivity Sketches

- Sketch: Simultaneously construct k independent connectivity sketches $\{M_1G, M_2G, \ldots M_kG\}$.
- Run Algorithm in Sketch Space:
  - Use M_1G to find a spanning forest F_1 of G
  - Use $M_2G - M_2F_1 = M_2(G - F_1)$ to find F_2

Ingredient 2: Connectivity Sketches

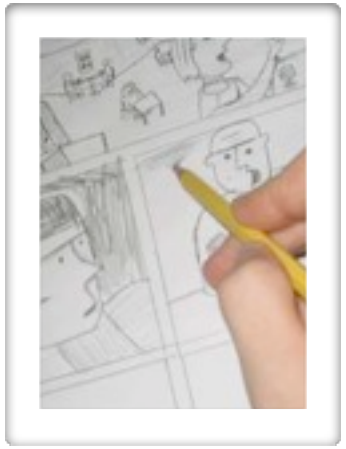
- Sketch: Simultaneously construct k independent connectivity sketches $\{M_1G, M_2G, \ldots M_kG\}$.
- Run Algorithm in Sketch Space:
  - Use M_1G to find a spanning forest F_1 of G
  - Use $M_2G-M_2F_1=M_2(G-F_1)$ to find F_2
  - Use $M_3G-M_3F_1-M_3F_2=M_3(G-F_1-F_2)$ to find F_3

Ingredient 2: Connectivity Sketches

- Sketch: Simultaneously construct k independent connectivity sketches $\{M_1G, M_2G, \dots M_kG\}$.
- Run Algorithm in Sketch Space:
  - Use M_1G to find a spanning forest F_1 of G
  - Use $M_2G-M_2F_1=M_2(G-F_1)$ to find F_2
  - Use $M_3G-M_3F_1-M_3F_2=M_3(G-F_1-F_2)$ to find F_3
  - etc.

Sketches Summary

- *Graph Sketches:* Linear projections that preserve structural graph properties. Results *parallelizable*, *streamable*, and *support deletions*.
- *Talk Results:* Projecting O(n)-dimensional neighborhoods to O(polylog n) dimensions while preserving connectivity and cuts.
- *Other Results:* Spanners, Bipartiteness, MST, Triangles, Matching, ...



And over to Part II…

Sağ olun!