

# Graph Stream Algorithms: *A Survey*



**Andrew McGregor**  
*University of Massachusetts Amherst*

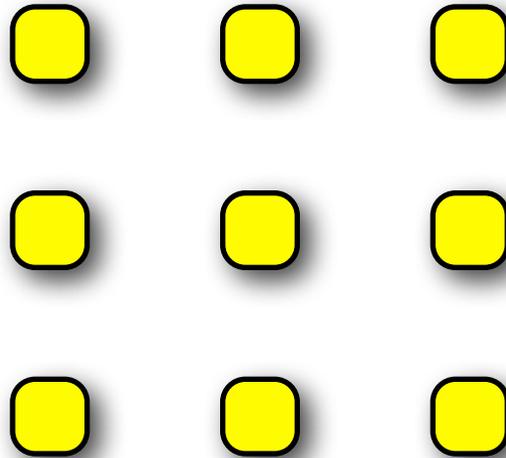
# Graph Stream Model

# Graph Stream Model

- Input: Sequence of edges  $(e_1, e_2 \dots)$  defines  $n$ -node graph  $G$ .

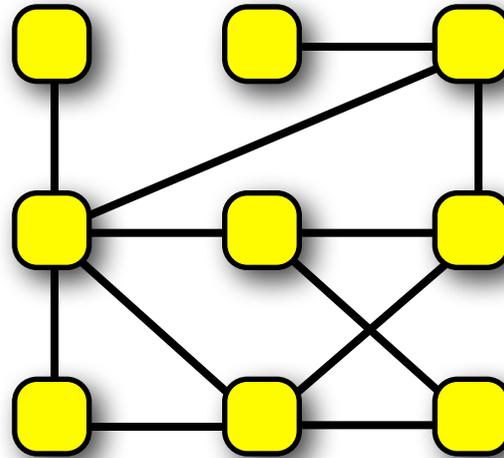
# Graph Stream Model

- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



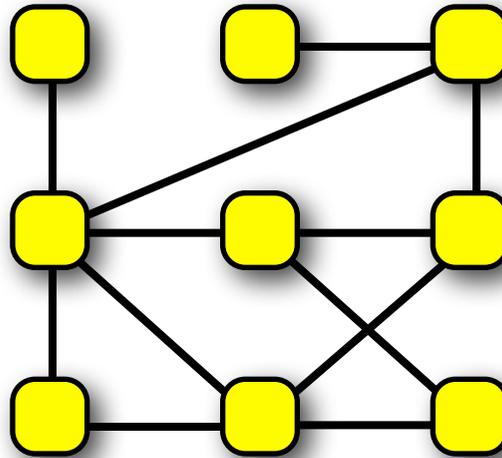
# Graph Stream Model

- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



# Graph Stream Model

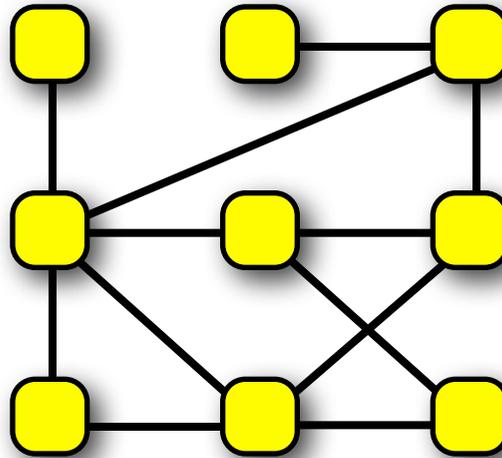
- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



- Goal: Compute properties of  $G$  without storing entire graph.

# Graph Stream Model

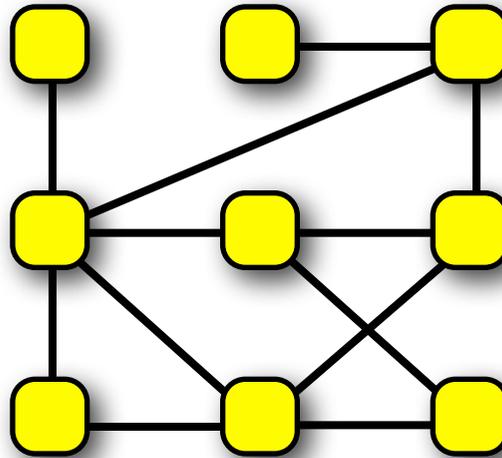
- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



- Goal: Compute properties of  $G$  without storing entire graph.
- Computational constraints:
  - i) **Limited working memory**, e.g.,  $O(n)$  rather than  $O(m)$

# Graph Stream Model

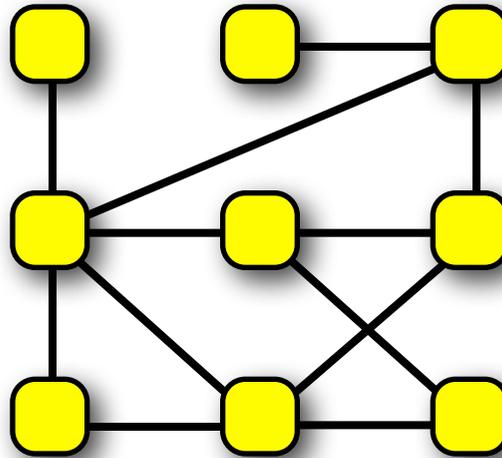
- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



- Goal: Compute properties of  $G$  without storing entire graph.
- Computational constraints:
  - i) Limited working memory, e.g.,  $O(n)$  rather than  $O(m)$
  - ii) Access data sequentially

# Graph Stream Model

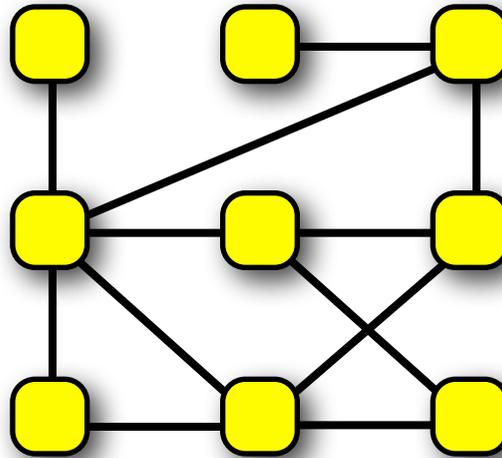
- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



- Goal: Compute properties of  $G$  without storing entire graph.
- Computational constraints:
  - i) Limited working memory, e.g.,  $O(n)$  rather than  $O(m)$
  - ii) Access data sequentially
  - iii) Process each element quickly

# Computational Graph Stream Model

- Input: Sequence of edges ( $e_1, e_2 \dots$ ) defines  $n$ -node graph  $G$ .



- Goal: Compute properties of  $G$  without storing entire graph.
- Computational constraints:
  - i) Limited working memory, e.g.,  $O(n)$  rather than  $O(m)$
  - ii) Access data sequentially
  - iii) Process each element quickly

# Motivation

# Motivation

- *Traditional stream applications:* Network monitoring, reading large data sets from disk, aggregation of sensor readings...

# Motivation

- *Traditional stream applications:* Network monitoring, reading large data sets from disk, aggregation of sensor readings...
- *Interesting theoretical questions:* How can we summarize graphs? Is there a notion of dimensionality reduction? What types of sampling is possible? Connections to compressed sensing, communication complexity, approximation, embeddings, ...

# Motivation

- Traditional stream applications: Network monitoring, reading large data sets from disk, aggregation of sensor readings...
- Interesting theoretical questions: How can we summarize graphs? Is there a notion of dimensionality reduction? What types of sampling is possible? Connections to compressed sensing, communication complexity, approximation, embeddings, ...
- Techniques have wider applications: E.g., distributed settings,

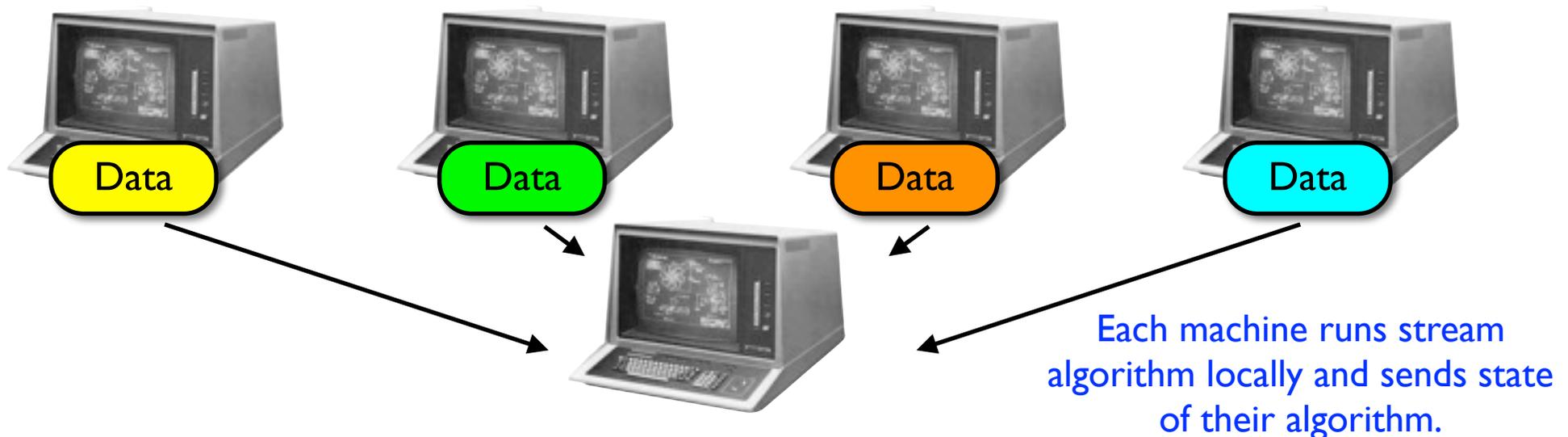
# Motivation

- Traditional stream applications: Network monitoring, reading large data sets from disk, aggregation of sensor readings...
- Interesting theoretical questions: How can we summarize graphs? Is there a notion of dimensionality reduction? What types of sampling is possible? Connections to compressed sensing, communication complexity, approximation, embeddings, ...
- Techniques have wider applications: E.g., distributed settings,



# Motivation

- Traditional stream applications: Network monitoring, reading large data sets from disk, aggregation of sensor readings...
- Interesting theoretical questions: How can we summarize graphs? Is there a notion of dimensionality reduction? What types of sampling is possible? Connections to compressed sensing, communication complexity, approximation, embeddings, ...
- Techniques have wider applications: E.g., distributed settings,



# Outline

# Outline

- *This Talk:*
  - **Algorithms:** Summarizing and computing on graph streams
  - **Extensions:** Sliding windows, extra passes, annotations etc.
  - **Future Directions:** Directed edges, ordering, stochastic graphs

# Outline

- *This Talk:*
  - **Algorithms:** Summarizing and computing on graph streams
  - **Extensions:** Sliding windows, extra passes, annotations etc.
  - **Future Directions:** Directed edges, ordering, stochastic graphs
- *Accompanying Survey:*
  - Includes all references and further details.
  - Feedback welcome...



<http://people.cs.umass.edu/~mcgregor/papers/13-graphsurvey.pdf>

**1. Algorithms**

**2. Extensions**

**3. Directions**

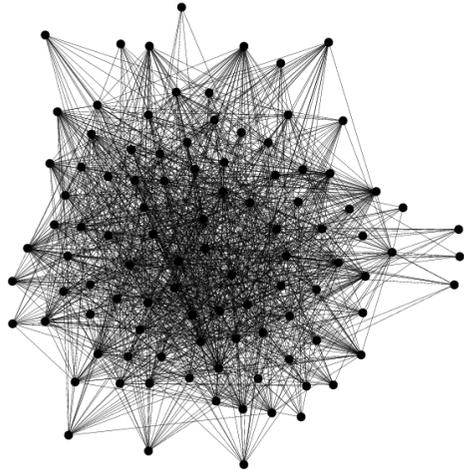
**1. Algorithms**

**2. Extensions**

**3. Directions**

# Sparsifiers & Cuts

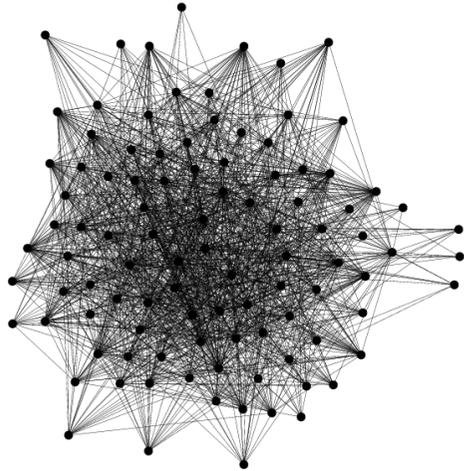
# Sparsifiers & Cuts



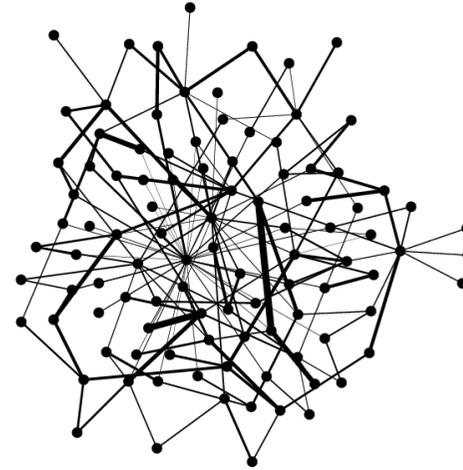
Original Graph G

- Sparsifiers: A subgraph  $H$  is a  $(1+\epsilon)$  sparsifier for  $G$  if the total weight of any cut is preserved up to a factor  $1+\epsilon$ .

# Sparsifiers & Cuts



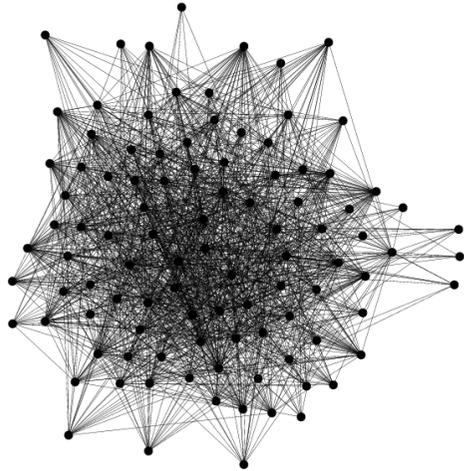
Original Graph G



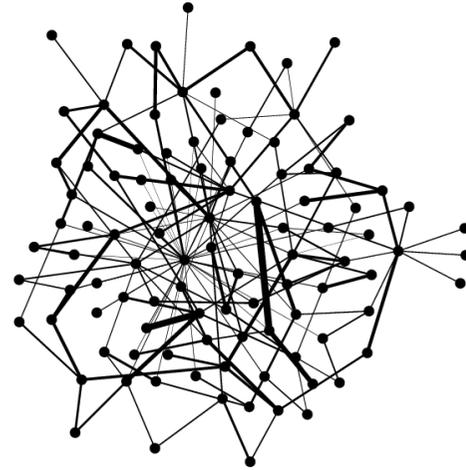
Sparsifier Graph H

- **Sparsifiers:** A subgraph H is a  $(1+\epsilon)$  sparsifier for G if the total weight of any cut is preserved up to a factor  $1+\epsilon$ .

# Sparsifiers & Cuts



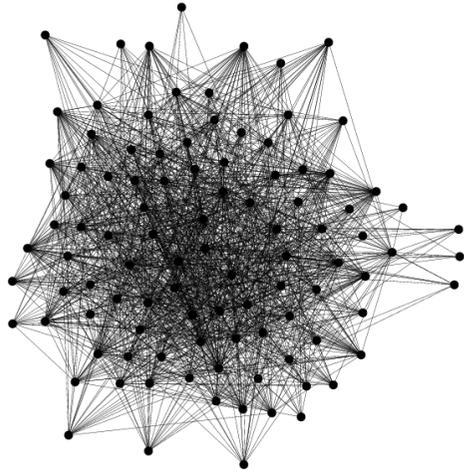
Original Graph G



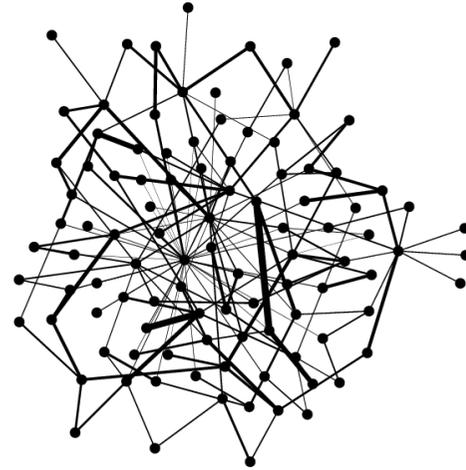
Sparsifier Graph H

- **Sparsifiers:** A subgraph H is a  $(1+\epsilon)$  sparsifier for G if the total weight of any cut is preserved up to a factor  $1+\epsilon$ .
- **Thm:** For any graph G there exists a  $(1+\epsilon)$  sparsifier with only  $O(\epsilon^{-2} n)$  edges. Can be constructed efficiently.

# Sparsifiers & Cuts



Original Graph G



Sparsifier Graph H

- **Sparsifiers:** A subgraph H is a  $(1+\epsilon)$  sparsifier for G if the total weight of any cut is preserved up to a factor  $1+\epsilon$ .
- **Thm:** For any graph G there exists a  $(1+\epsilon)$  sparsifier with only  $O(\epsilon^{-2} n)$  edges. Can be constructed efficiently.
- **Thm:** Can construct a  $(1+\epsilon)$ -sparsifier of a graph stream using  $O(\epsilon^{-2} n \text{ polylog } n)$  bits of space.

# Sparsifier Algorithm

# Sparsifier Algorithm

- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm

$E_1$

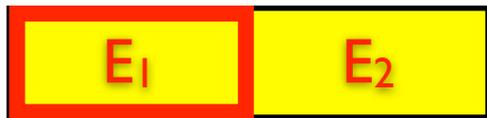
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm

$E_1$

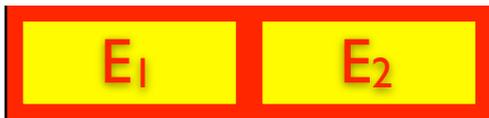
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



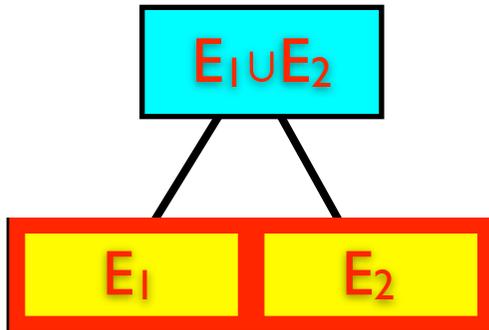
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



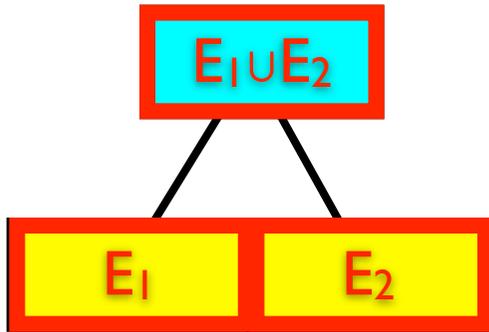
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



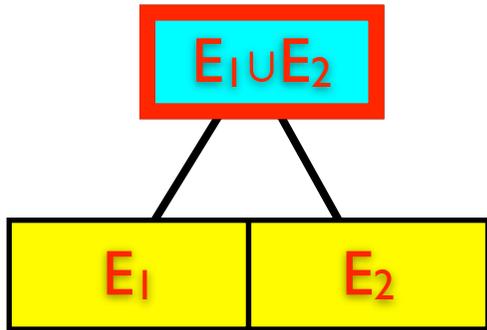
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



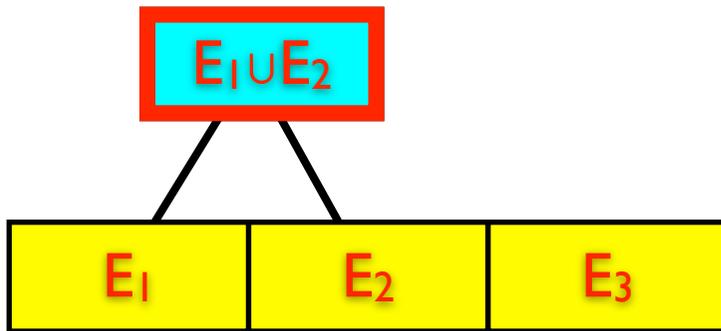
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



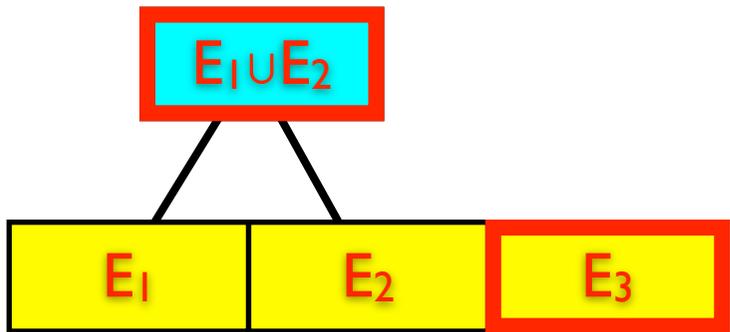
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



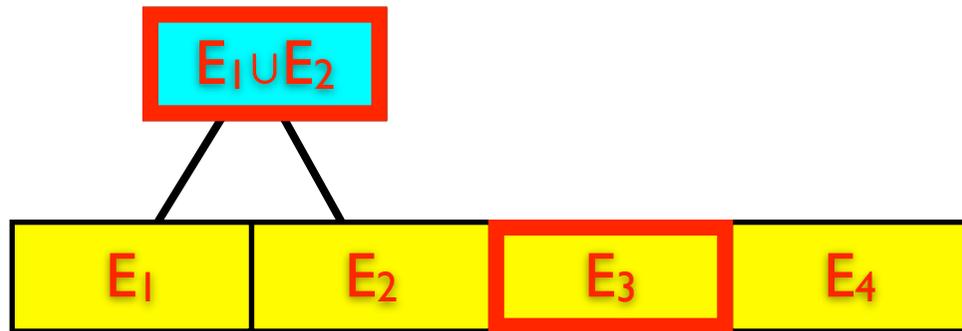
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



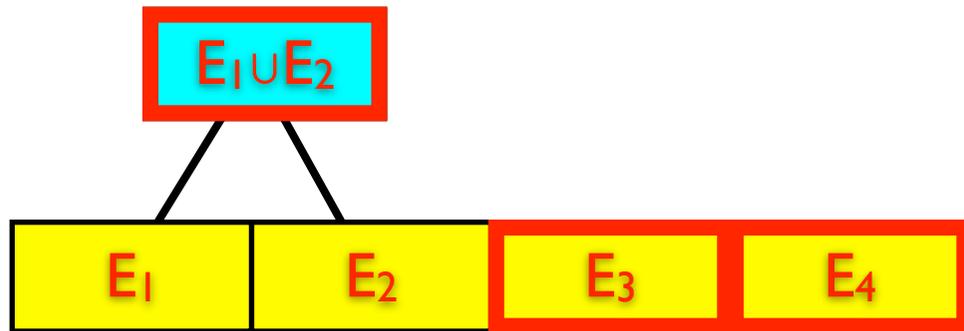
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



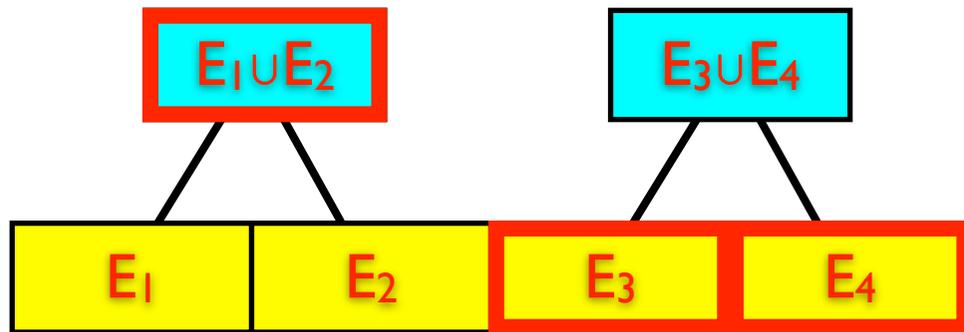
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



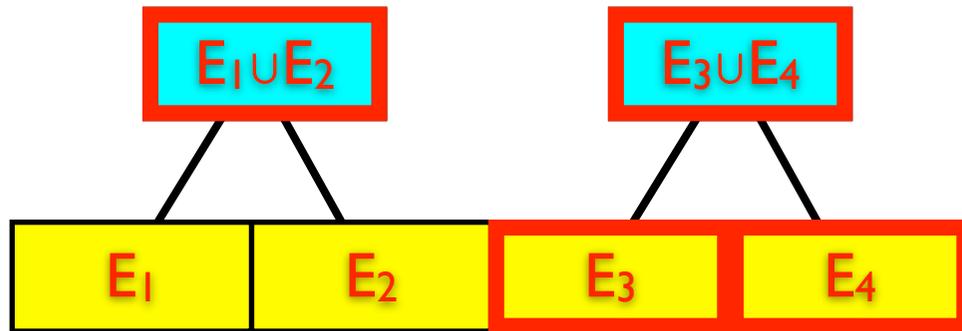
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



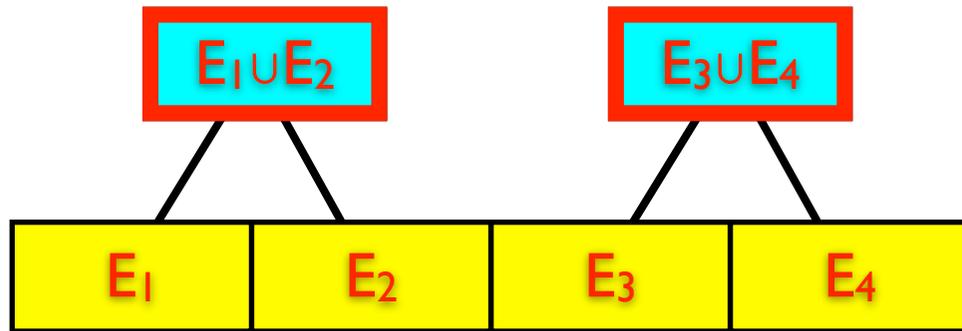
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



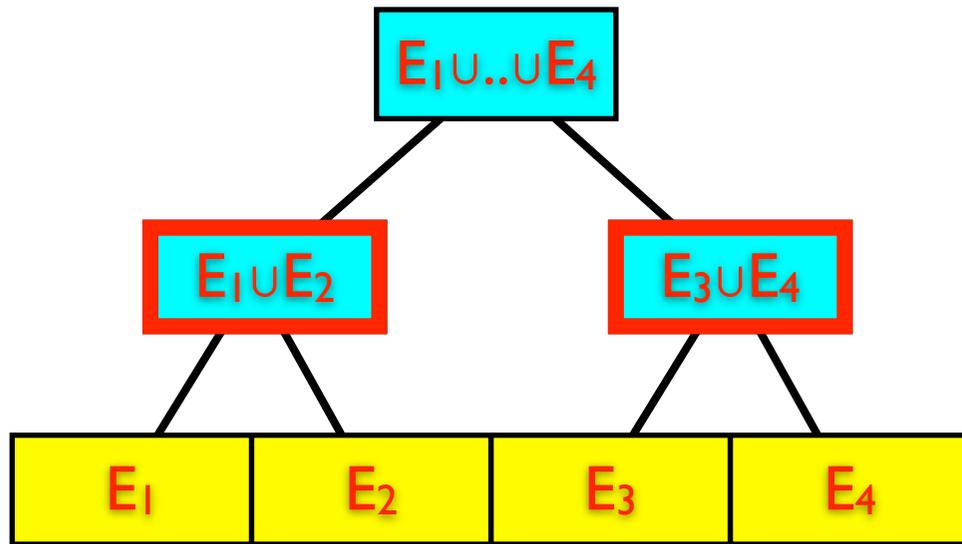
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



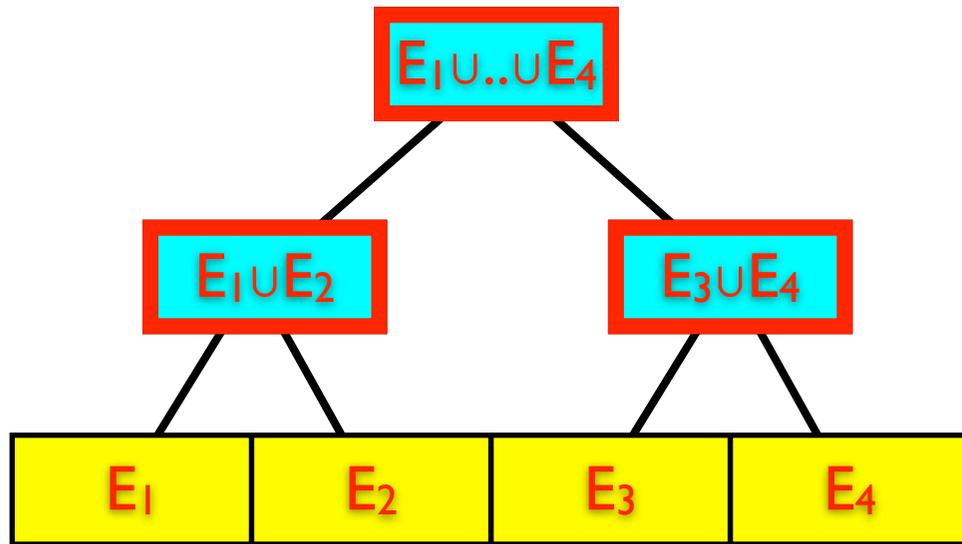
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



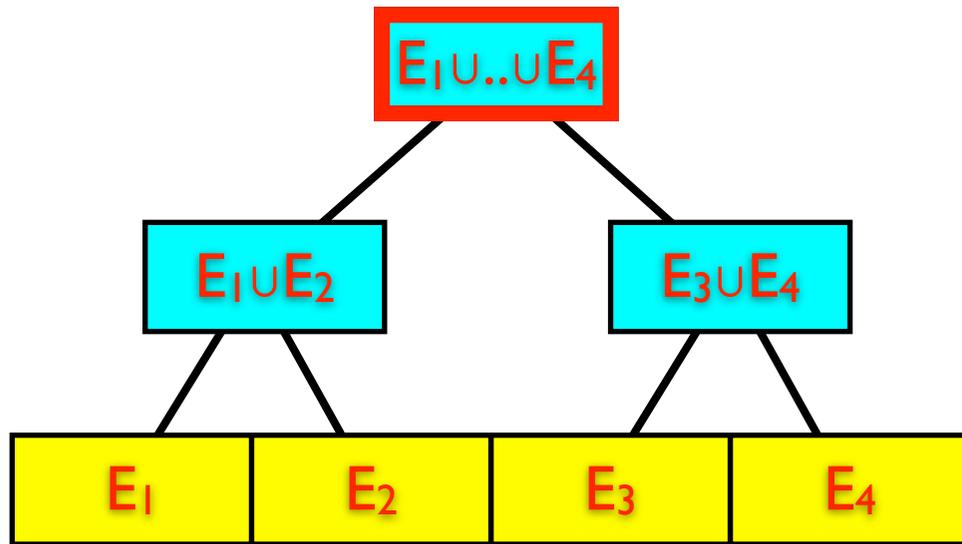
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



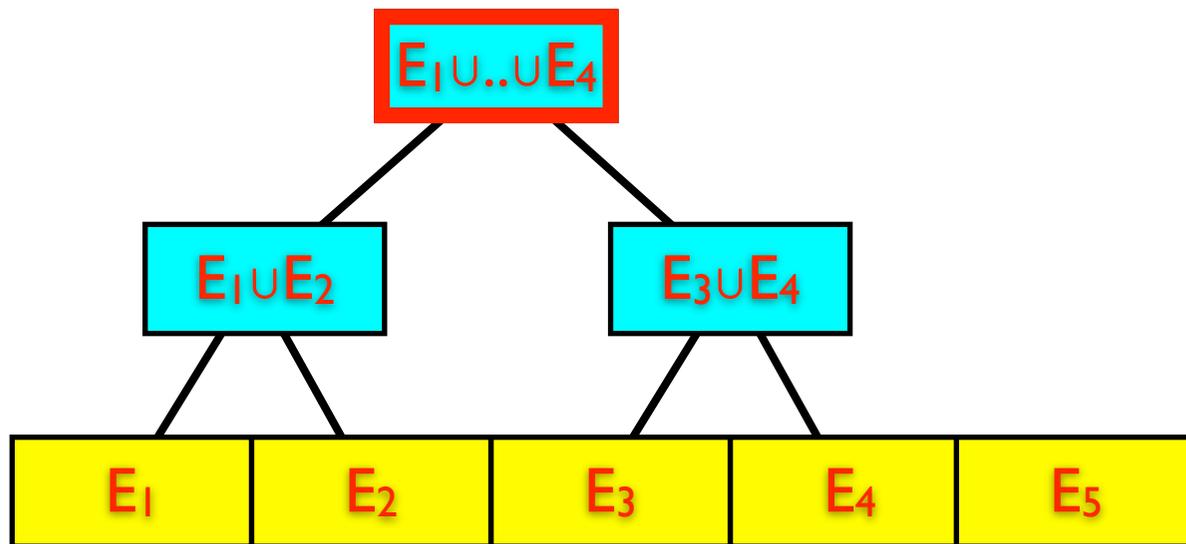
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



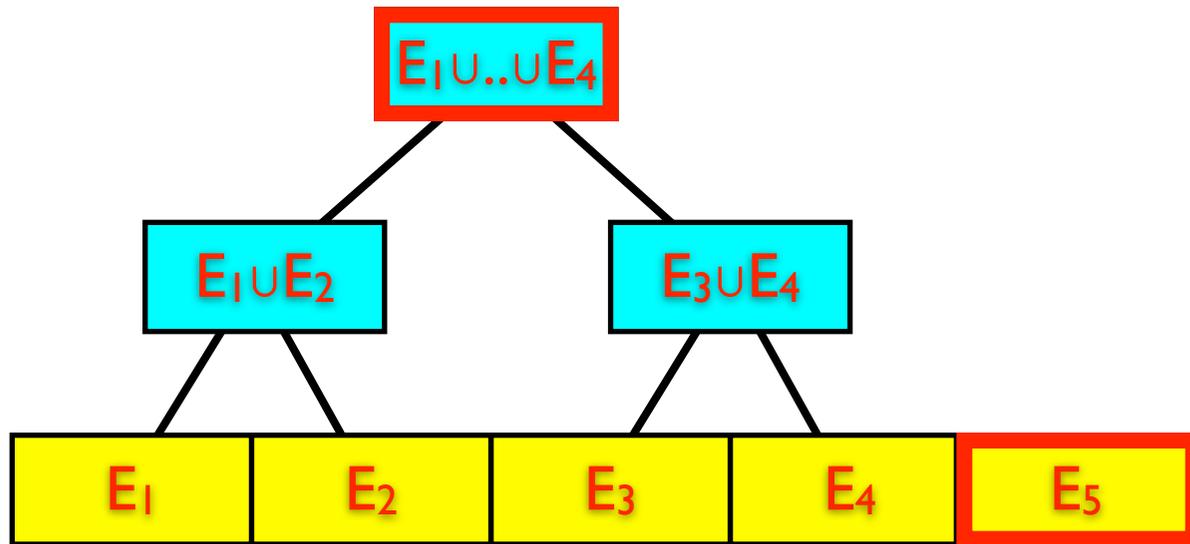
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



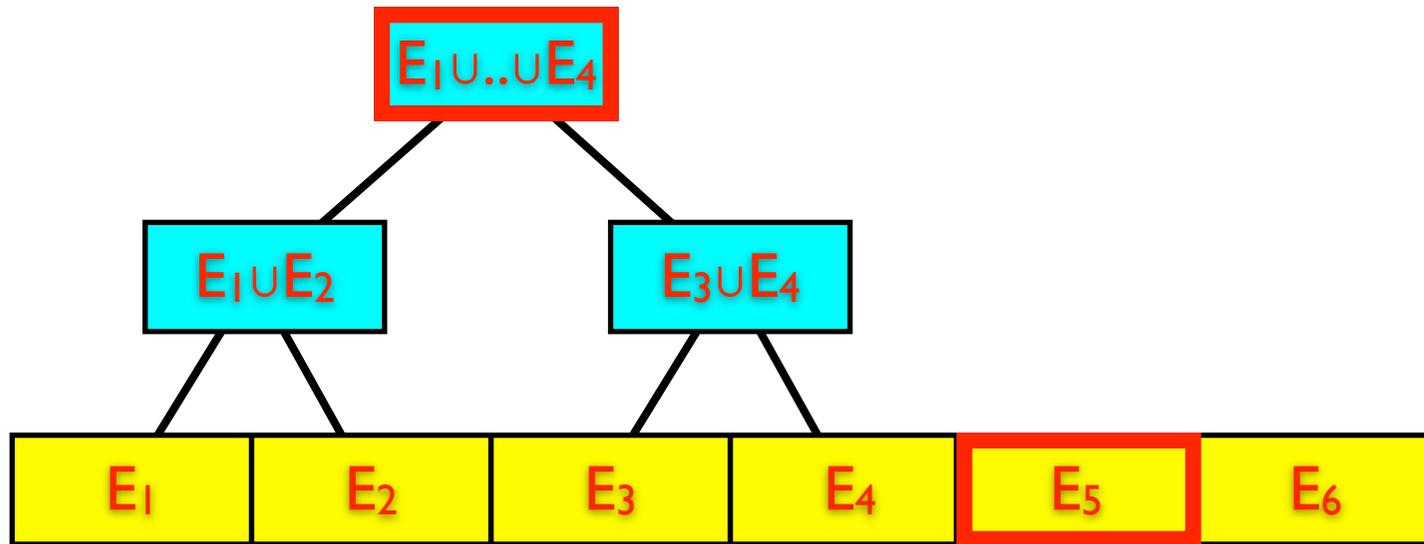
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



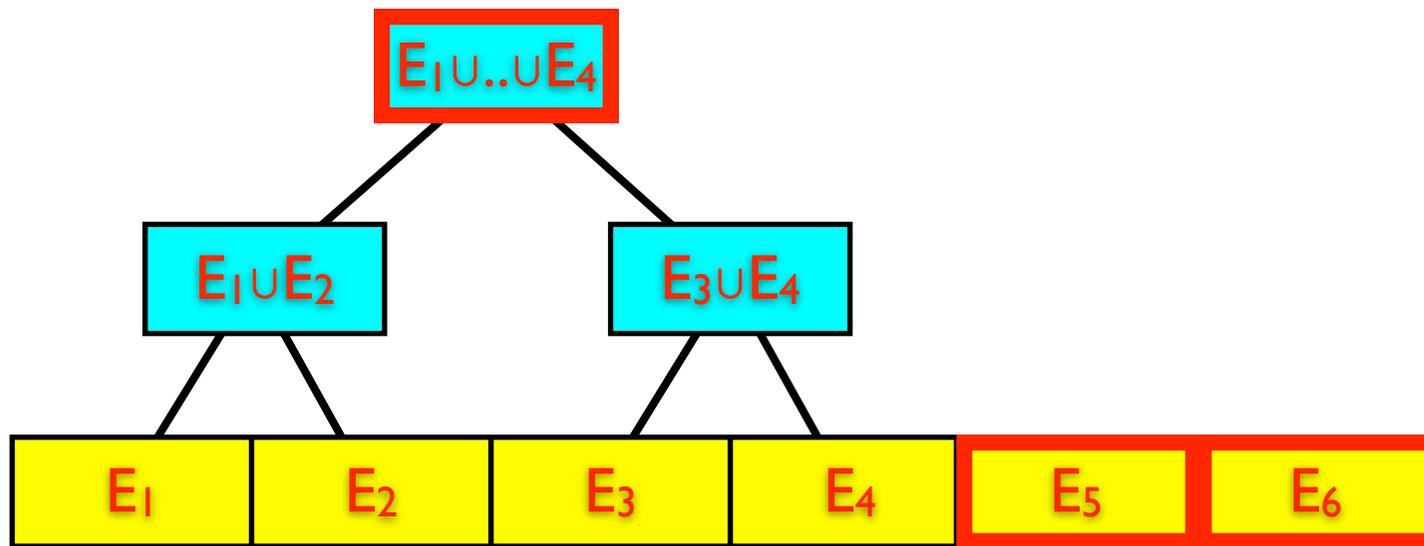
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



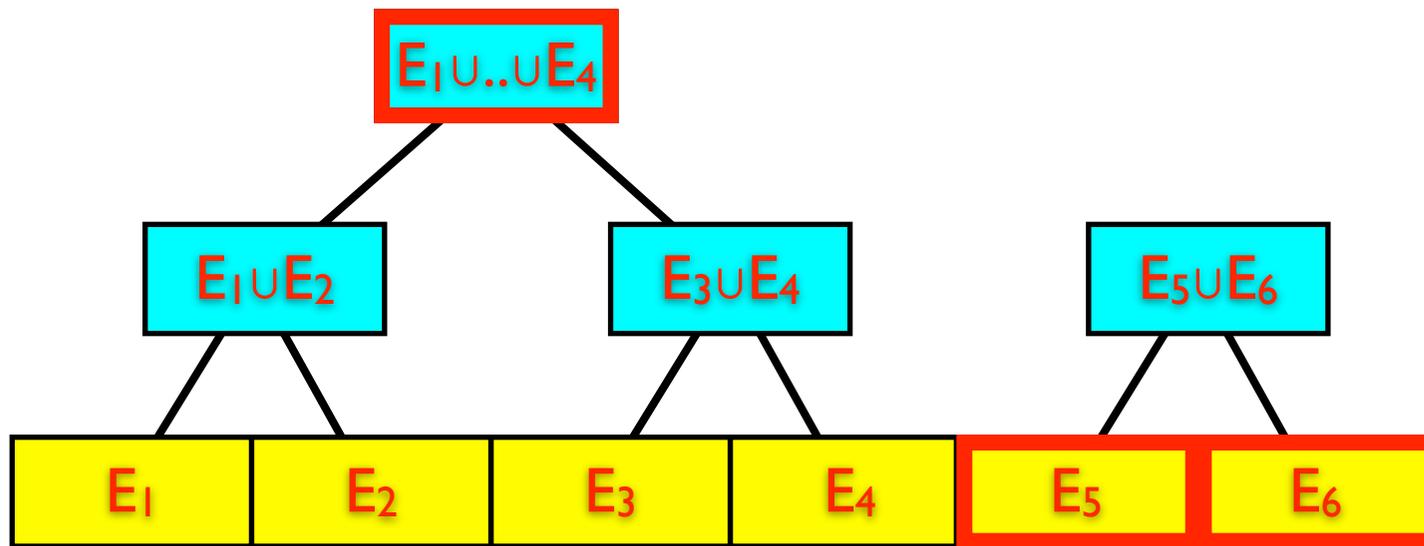
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



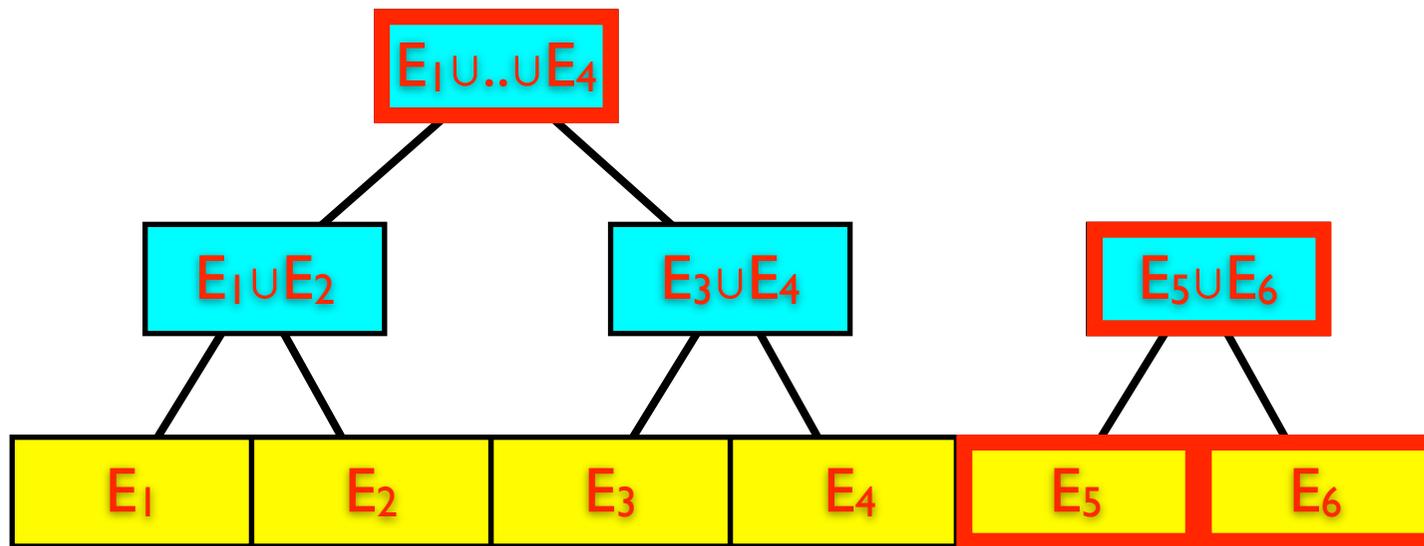
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



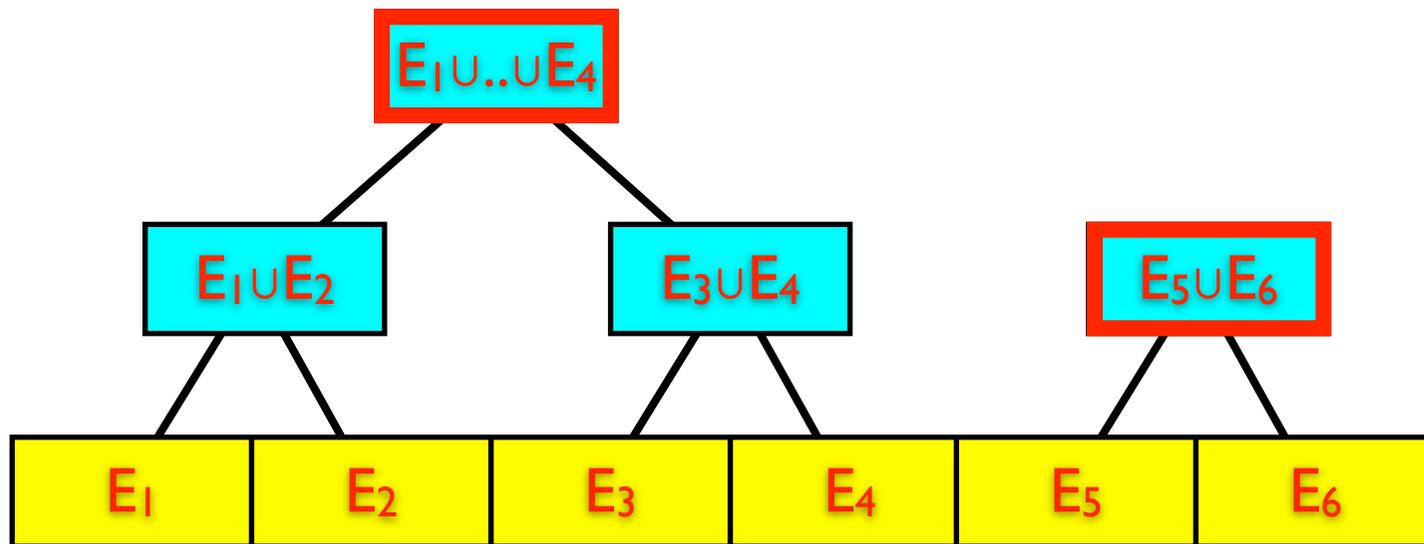
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



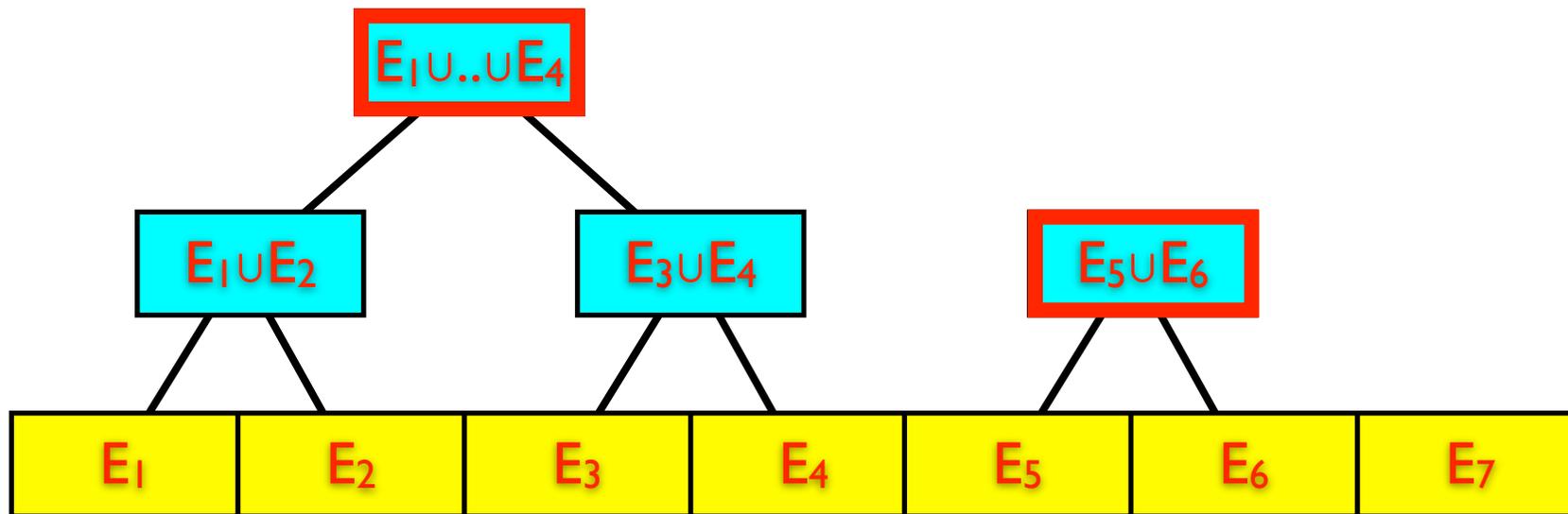
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



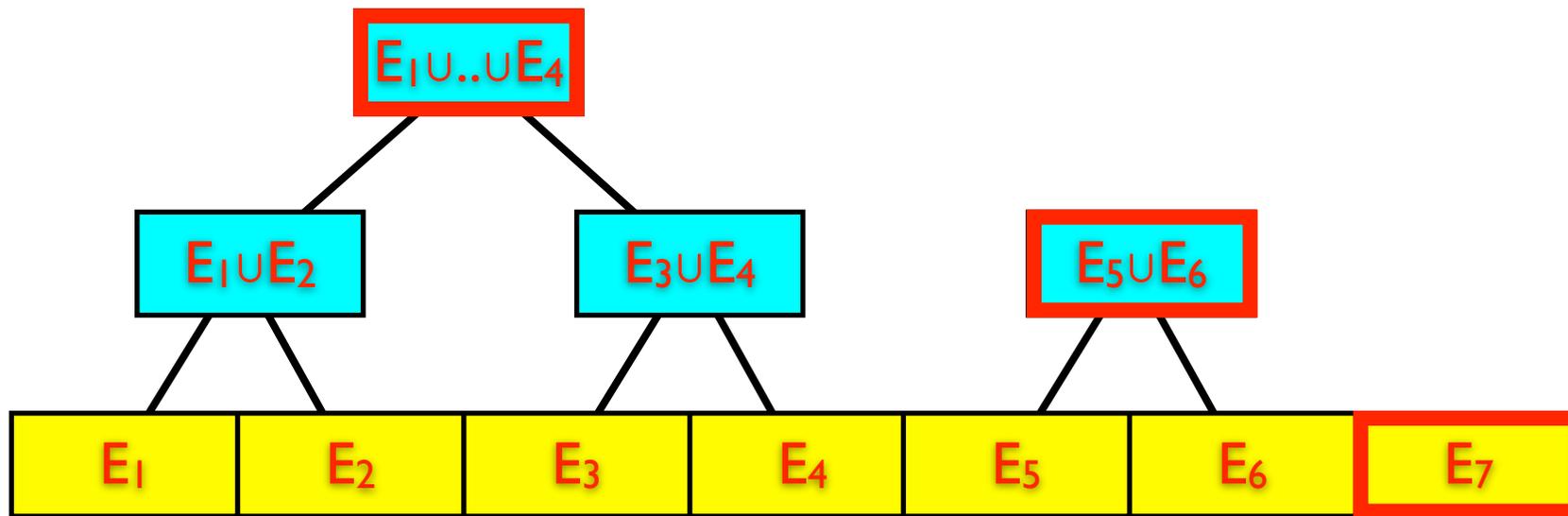
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



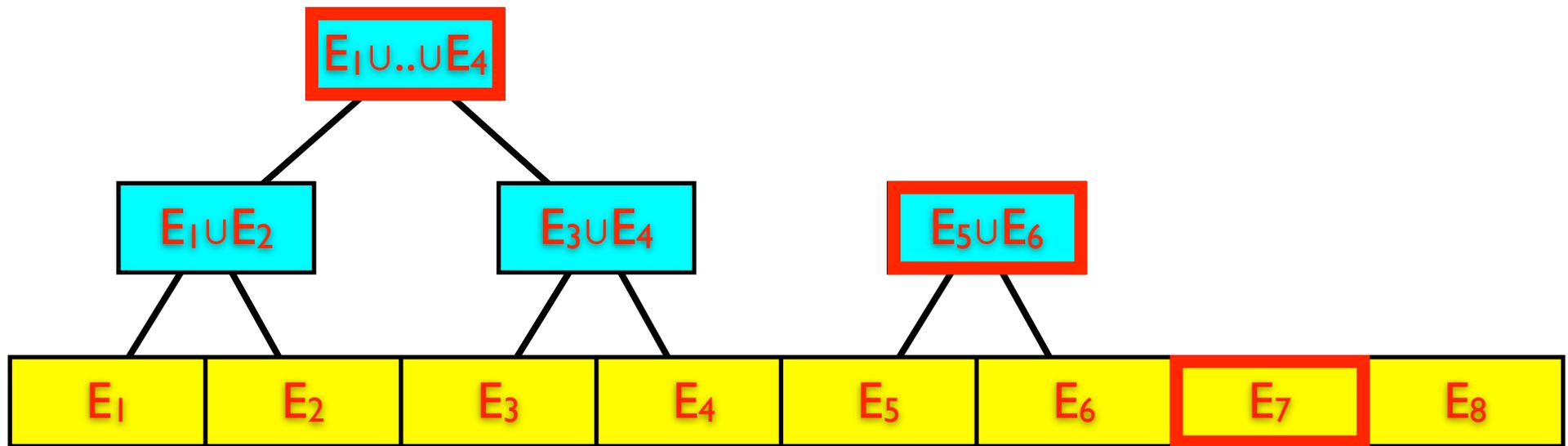
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



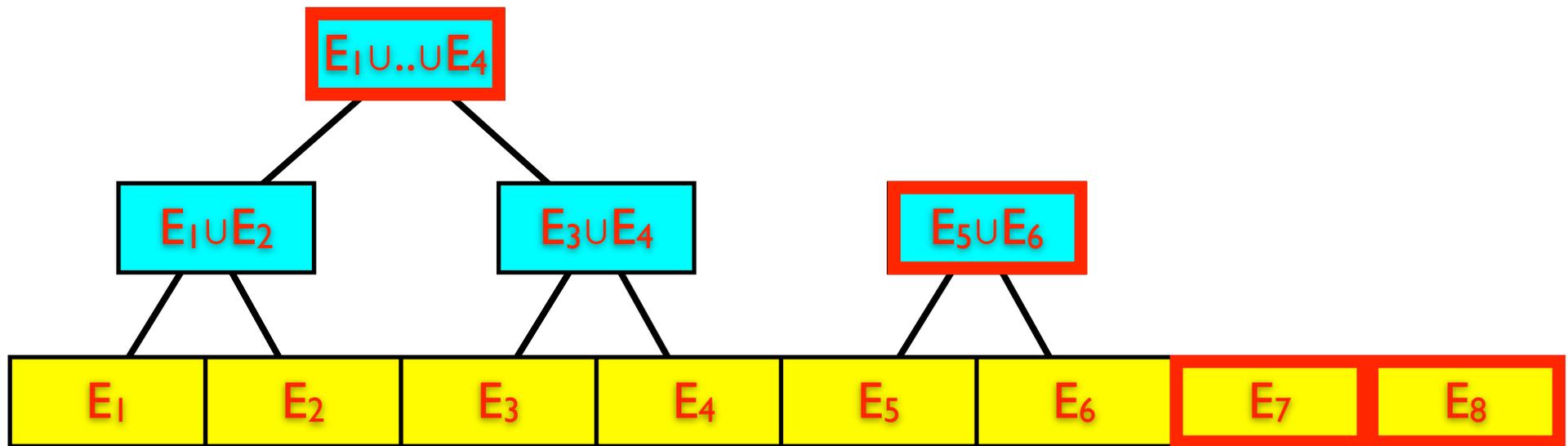
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



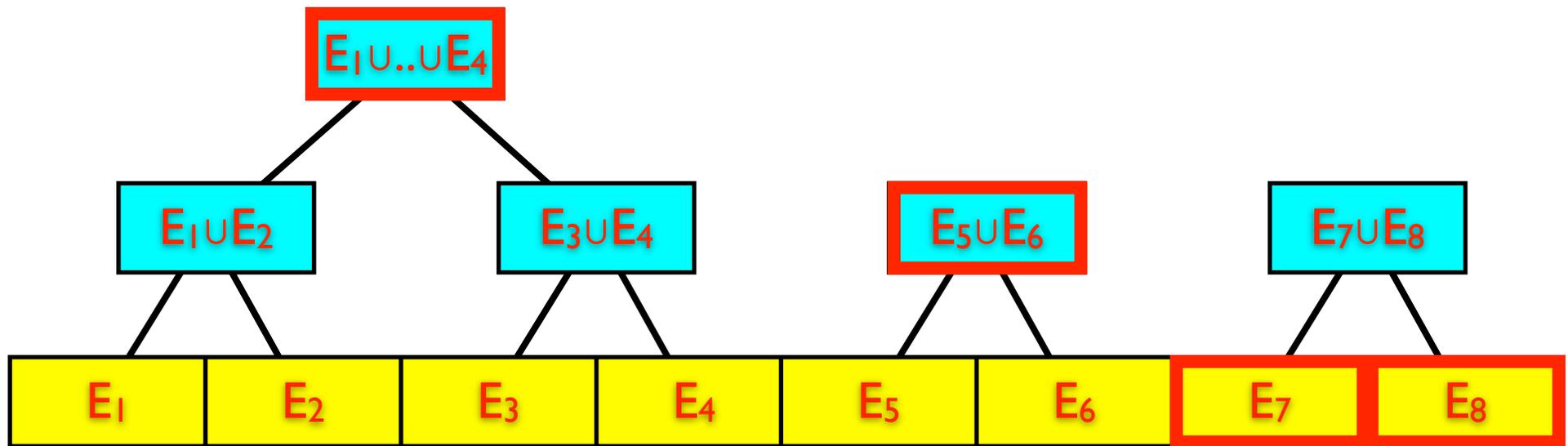
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



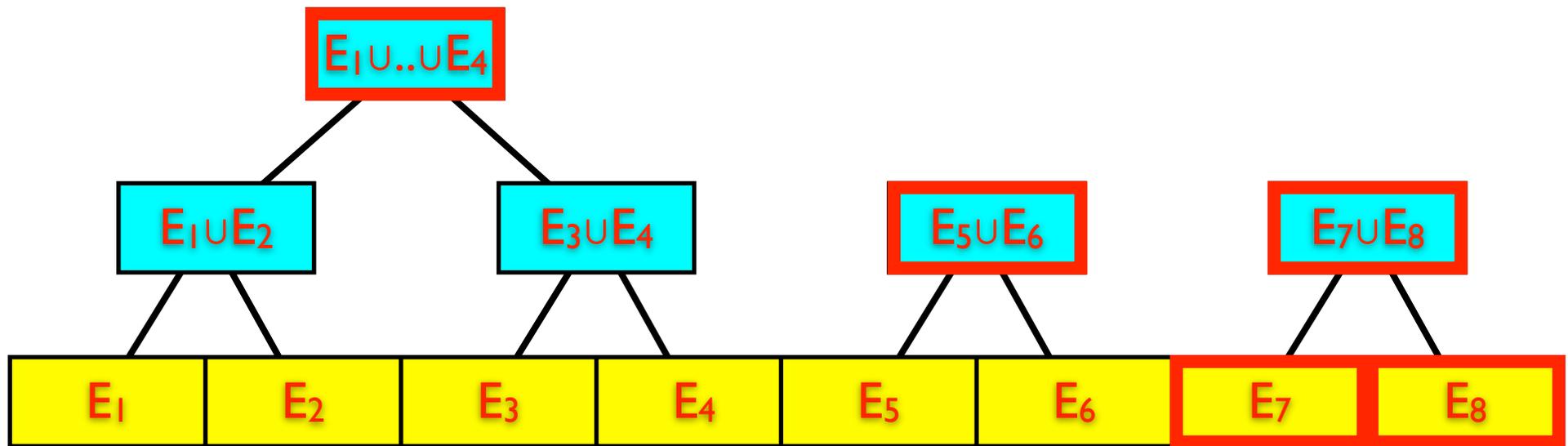
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



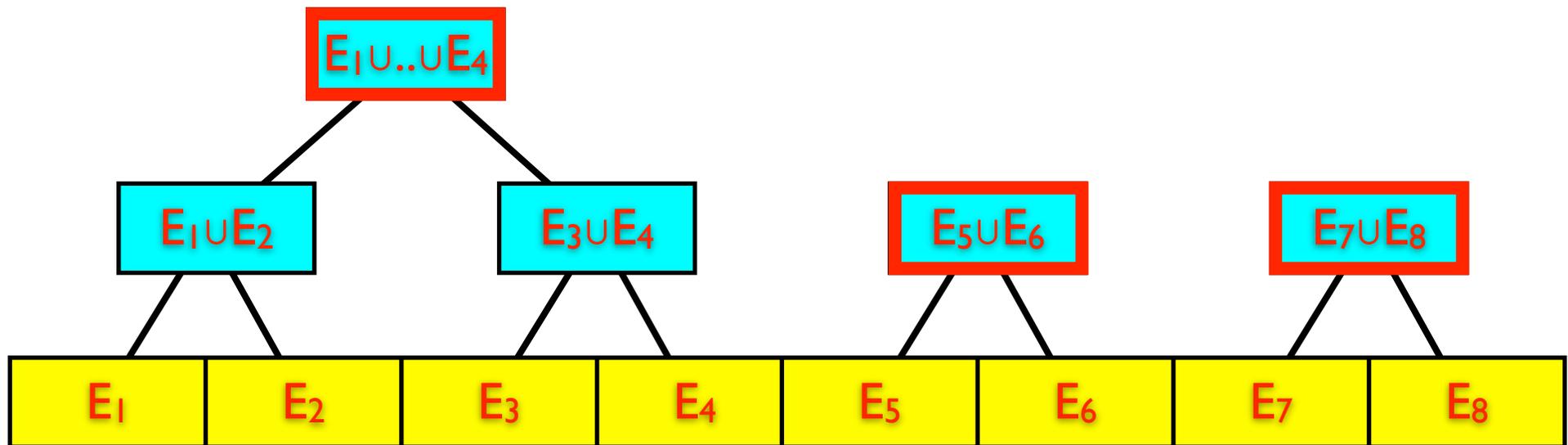
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



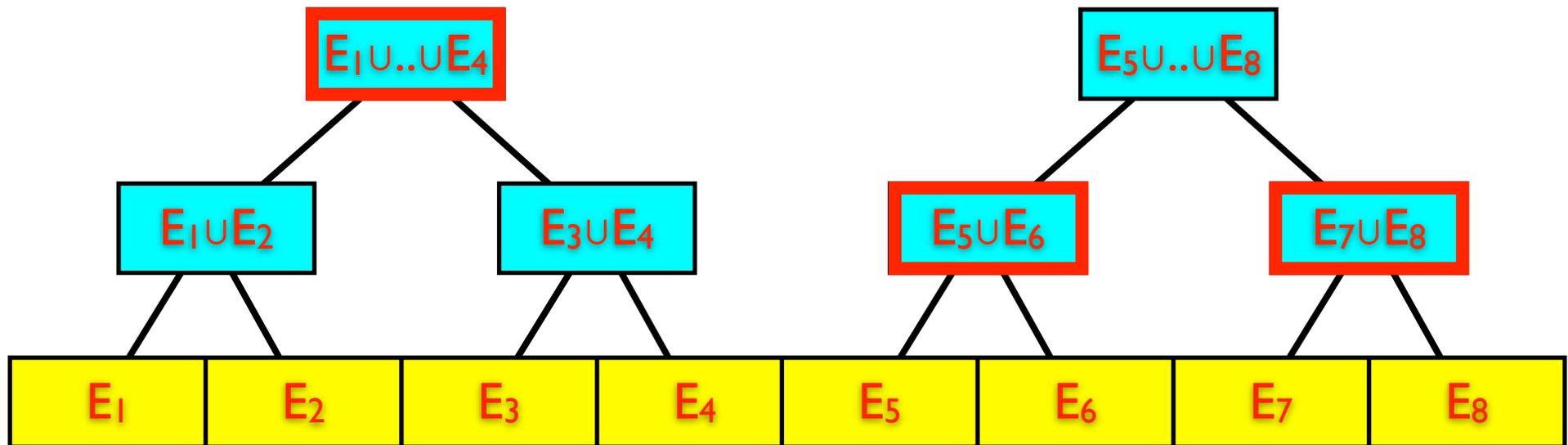
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



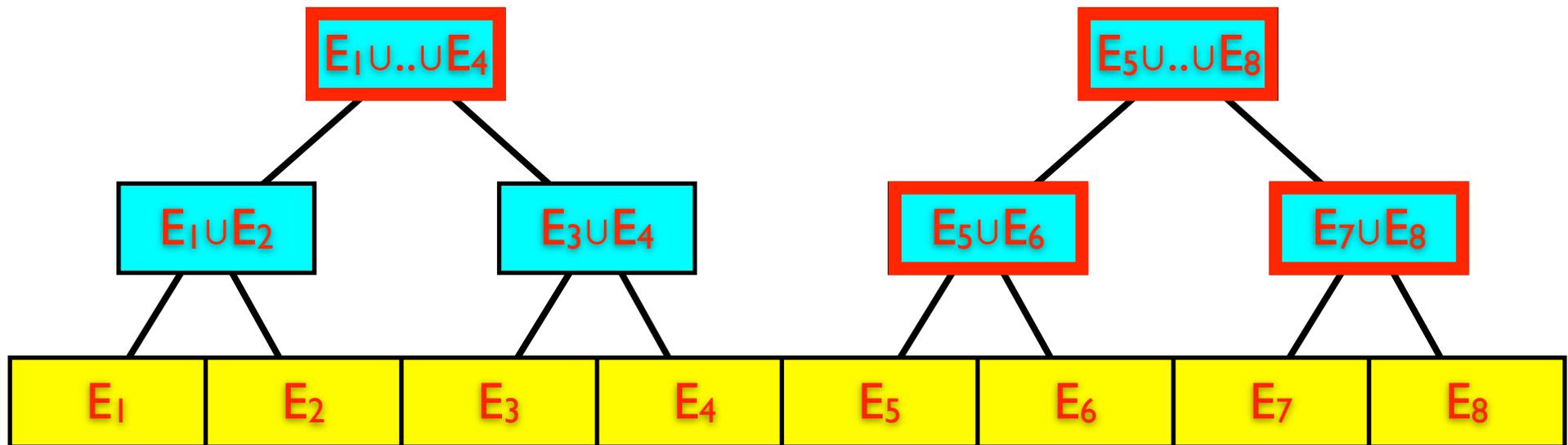
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



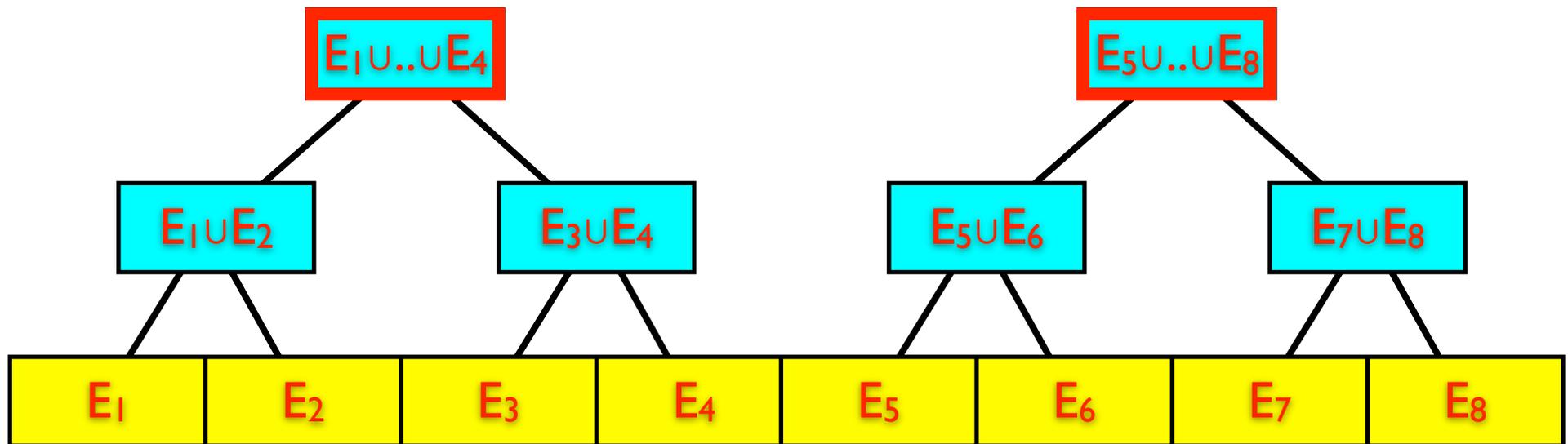
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



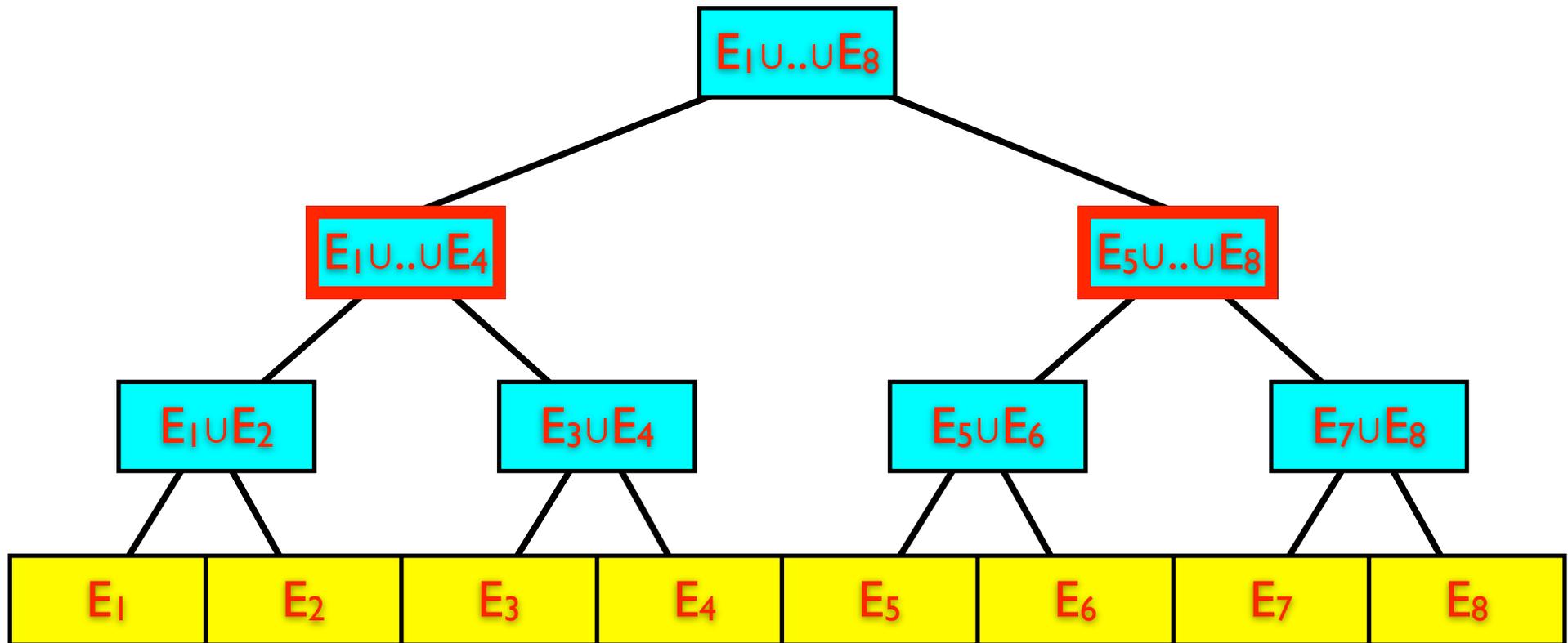
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



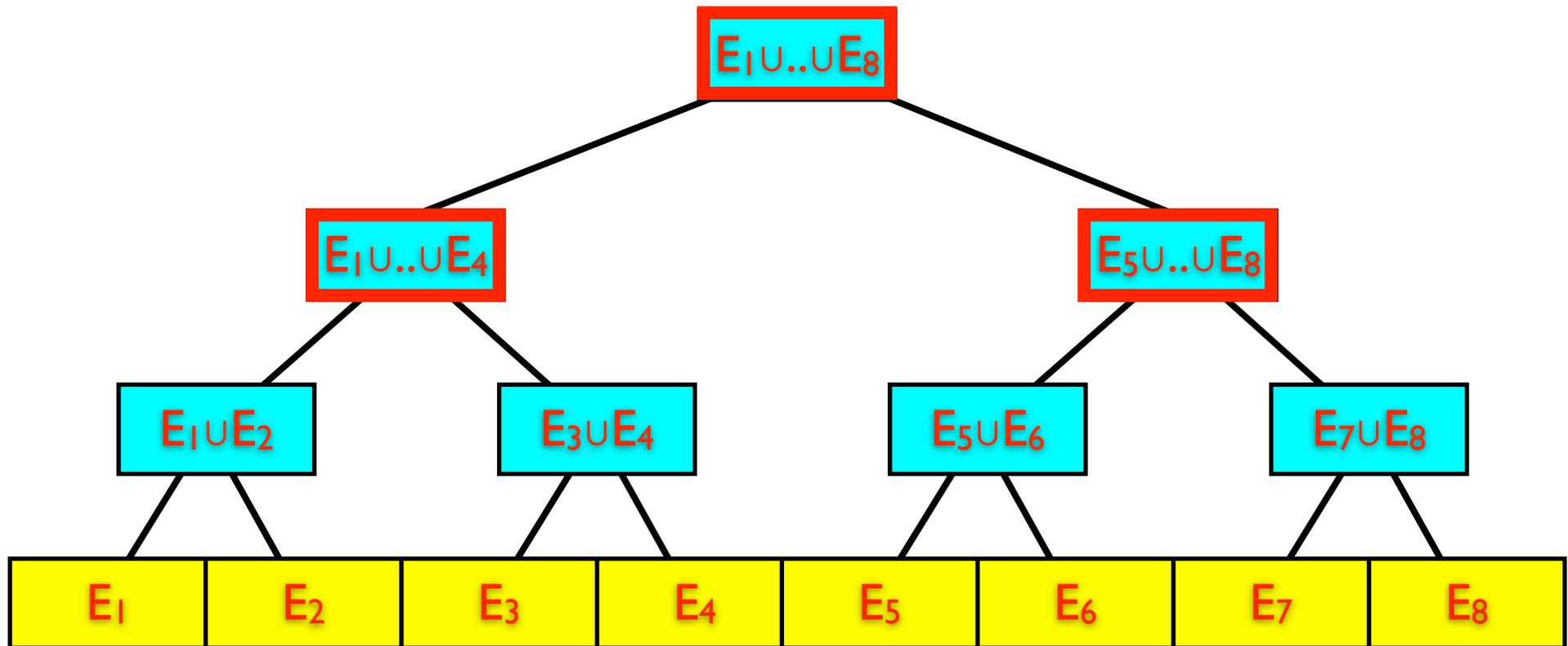
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



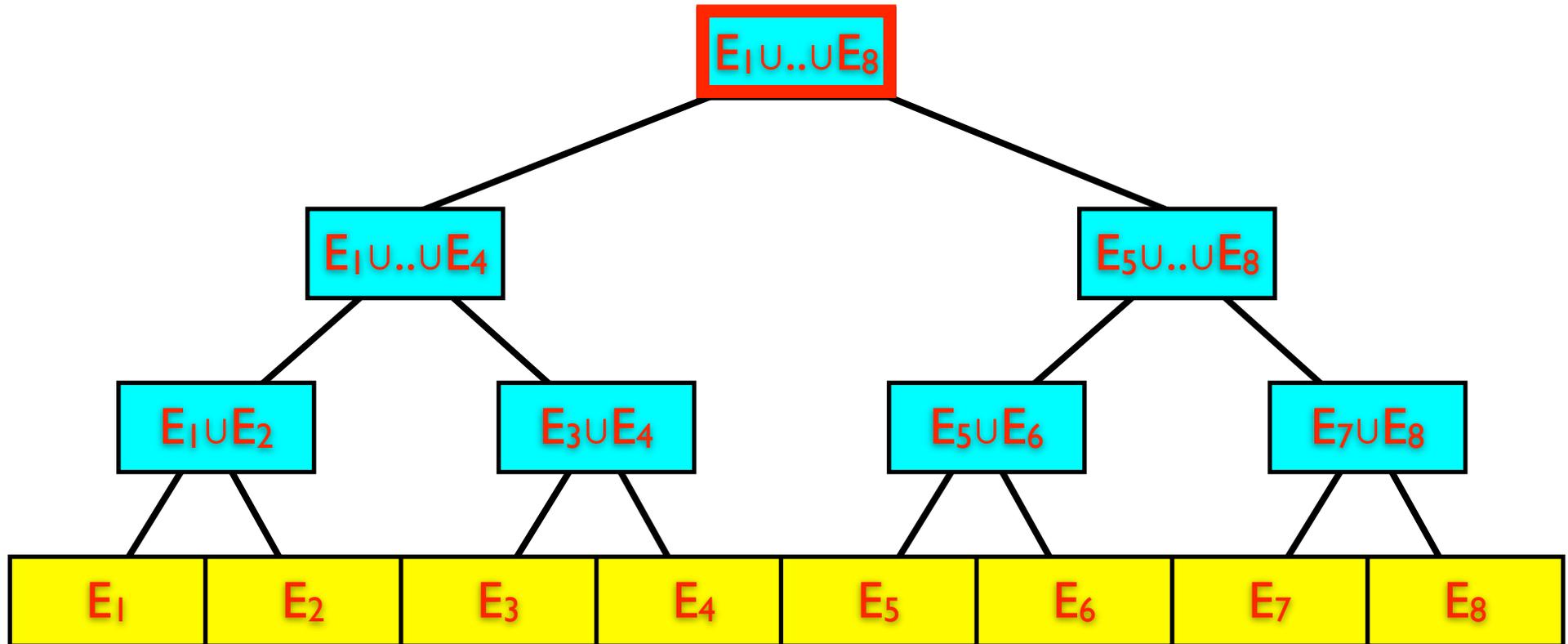
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



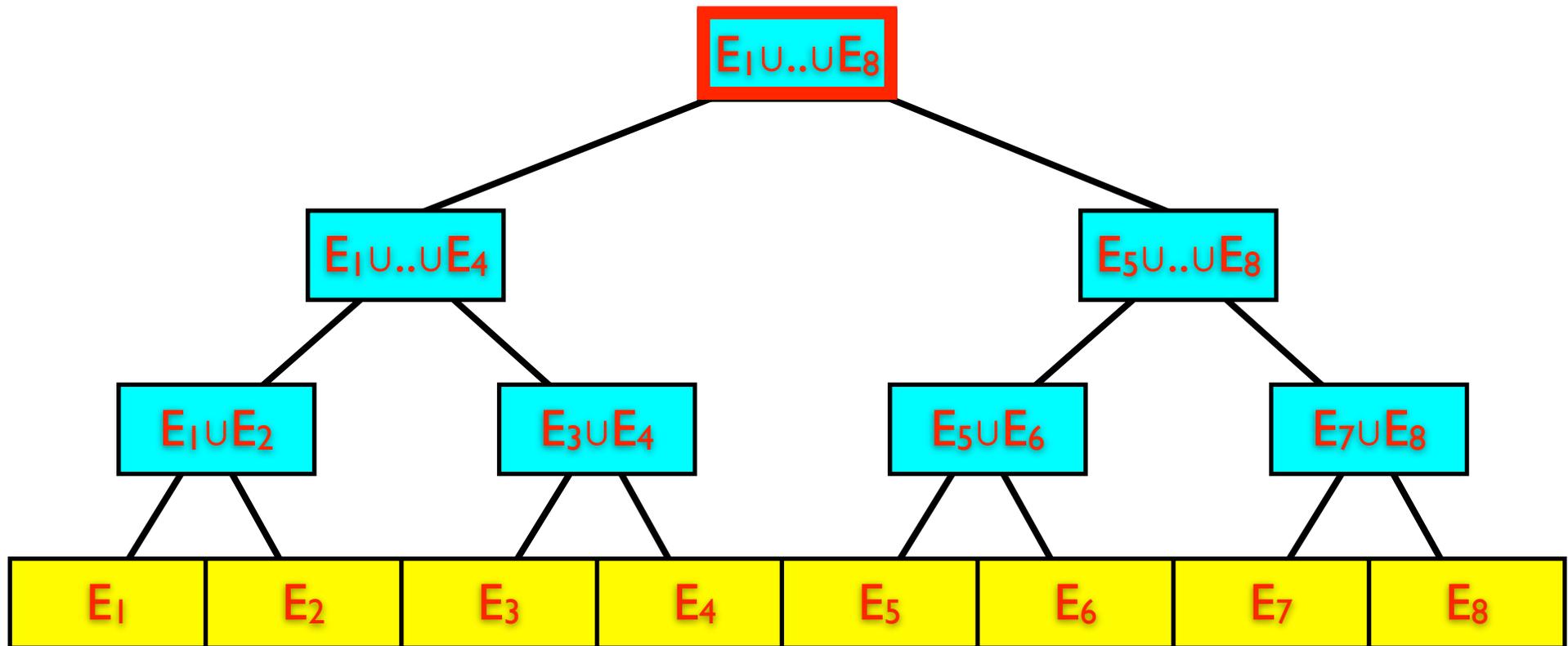
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



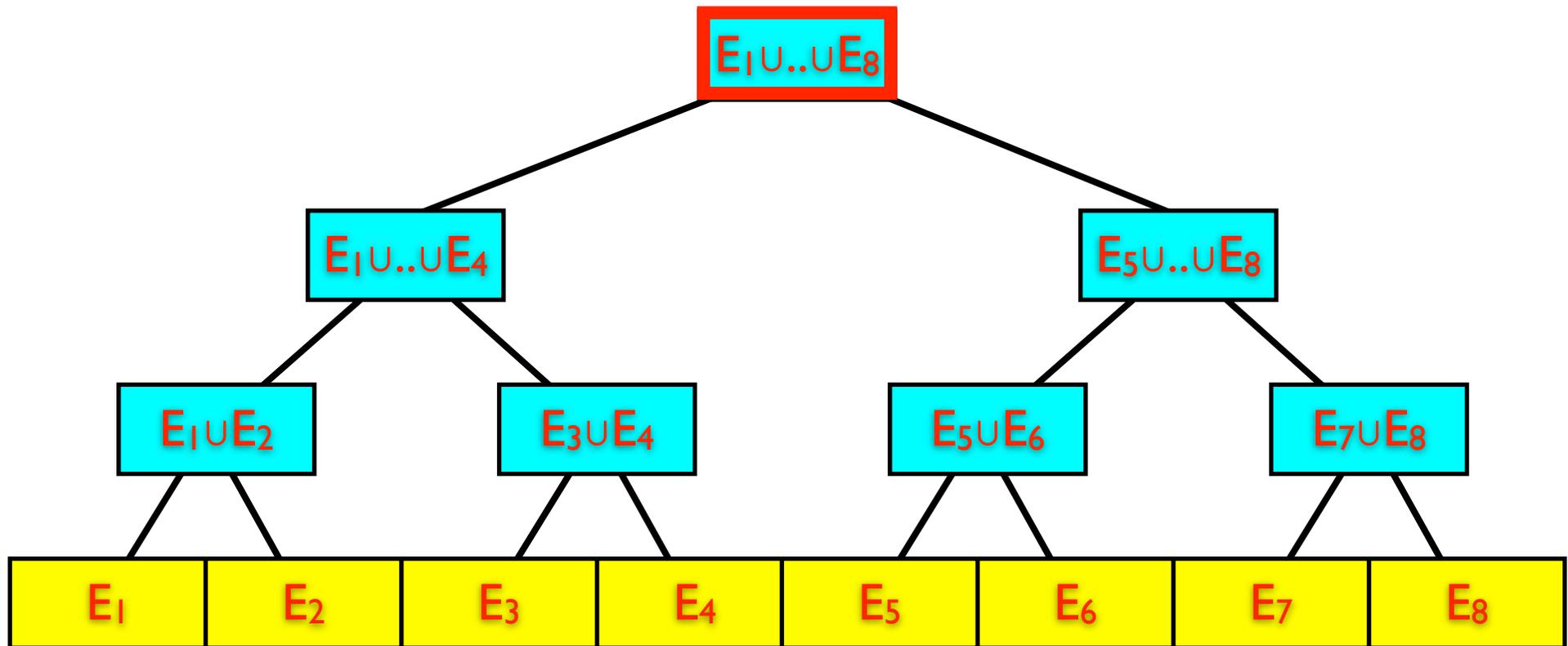
- Algorithm: Recursively re-sparsify using any “offline” algorithm.

# Sparsifier Algorithm



- **Algorithm:** Recursively re-sparsify using any “offline” algorithm.
- **Analysis:** Let  $d = O(\log n)$  be depth of the tree. Error of a final cut estimate is  $(1 + \epsilon)^d$  and we only store  $d$  sparsifiers simultaneously.

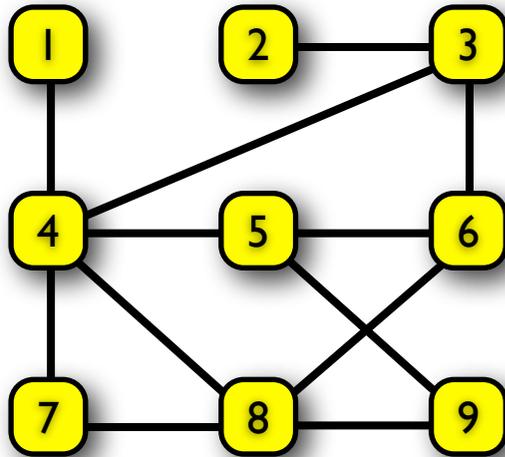
# Sparsifier Algorithm



- **Algorithm:** Recursively re-sparsify using any “offline” algorithm.
- **Analysis:** Let  $d = O(\log n)$  be depth of the tree. Error of a final cut estimate is  $(1 + \epsilon)^d$  and we only store  $d$  sparsifiers simultaneously.
- Results extend to constructing spectral sparsifiers.

# Spanners & Distances

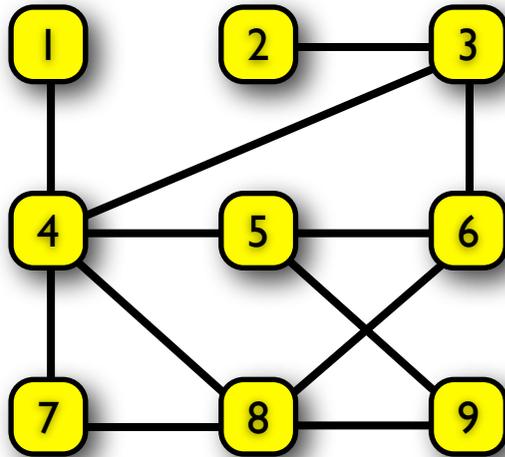
# Spanners & Distances



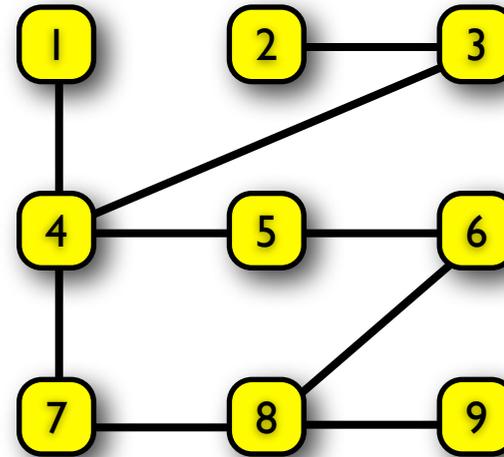
Original Graph G

- **Spanner**: A subgraph  $H$  is a  **$k$ -spanner** for  $G$  if all graph distances are preserved up to a factor  $k$ .

# Spanners & Distances



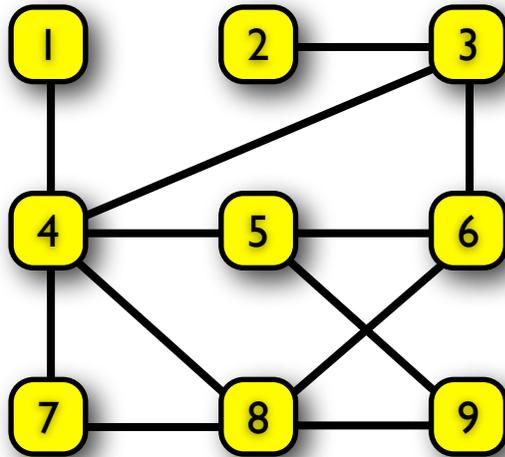
Original Graph G



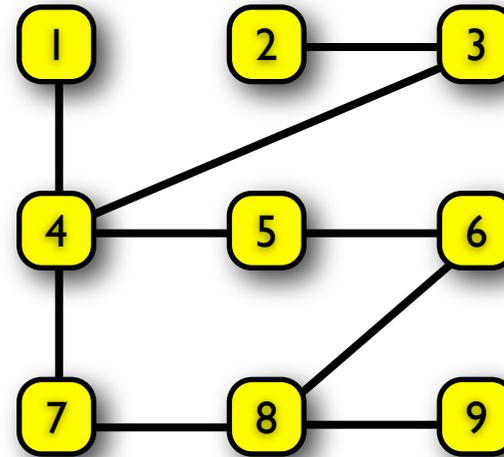
Spanner Graph H

- **Spanner**: A subgraph H is a **k-spanner** for G if all graph distances are preserved up to a factor k.

# Spanners & Distances



Original Graph G

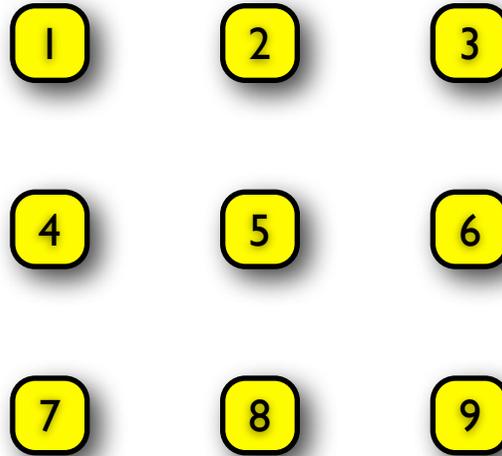


Spanner Graph H

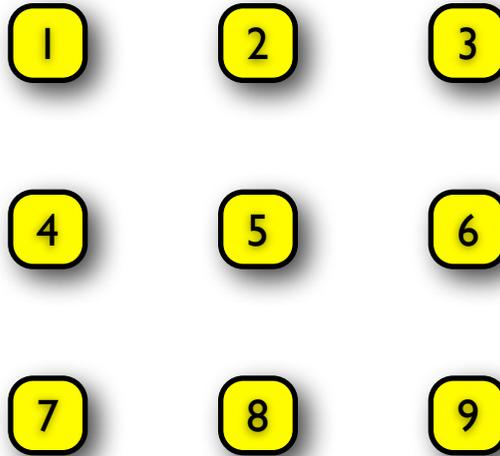
- **Spanner**: A subgraph H is a **k-spanner** for G if all graph distances are preserved up to a factor k.
- **Thm**: There is a  $O(n^{1+1/t})$  space stream algorithm that constructs a  $(2t-1)$ -spanner.

# Spanners Algorithm

# Spanners Algorithm

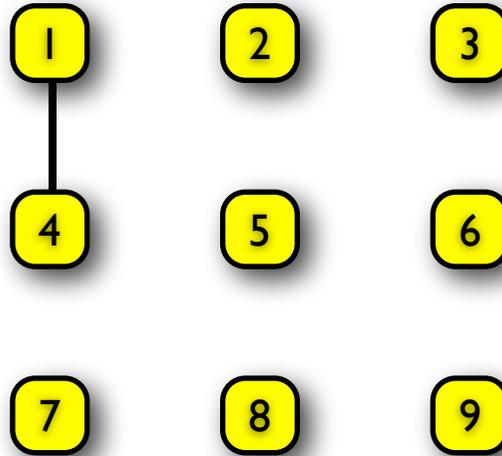


# Spanners Algorithm



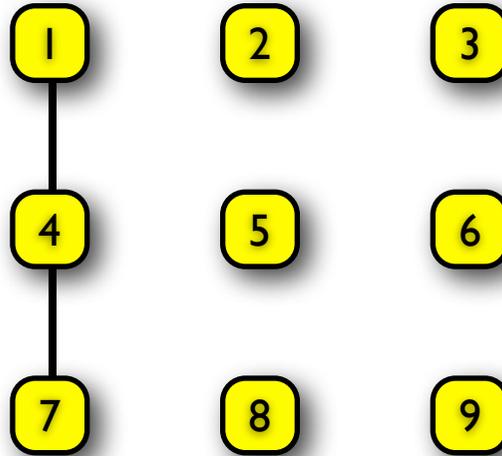
- Algorithm: Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



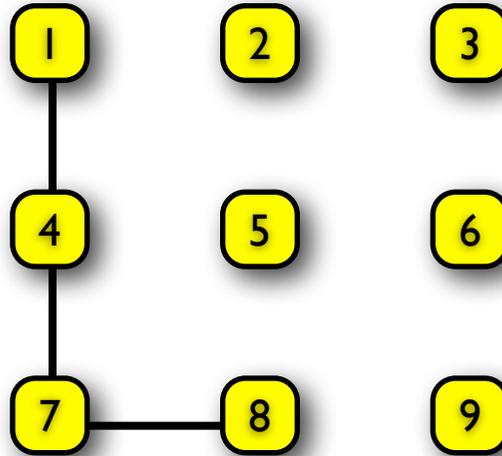
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



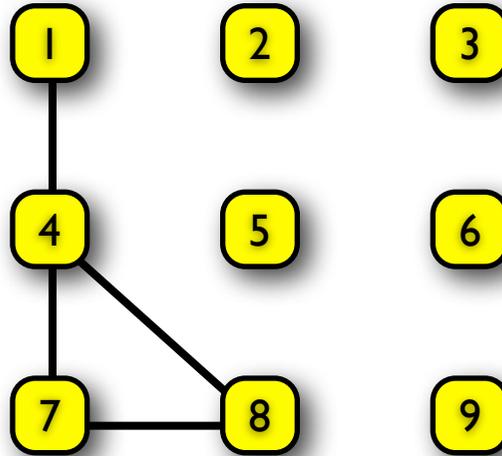
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



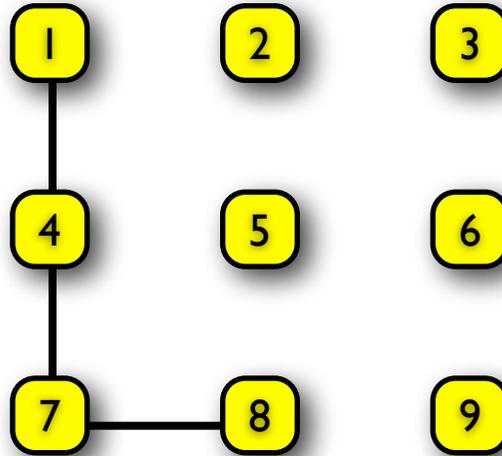
- Algorithm: Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



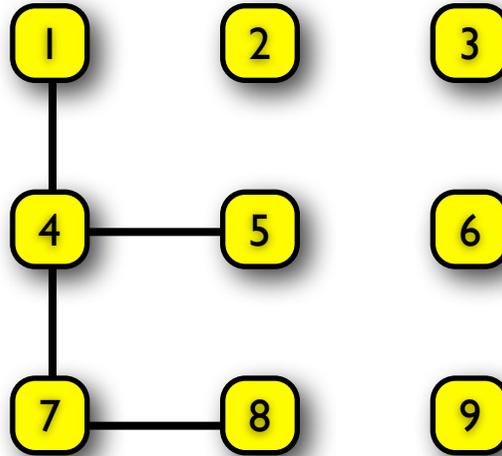
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



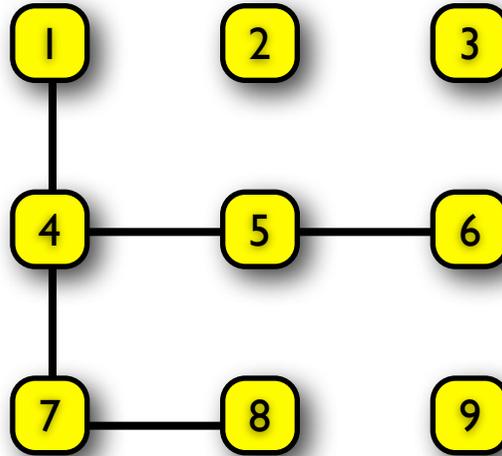
- Algorithm: Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



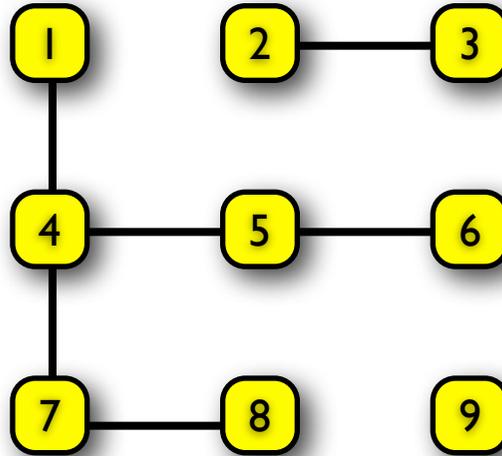
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



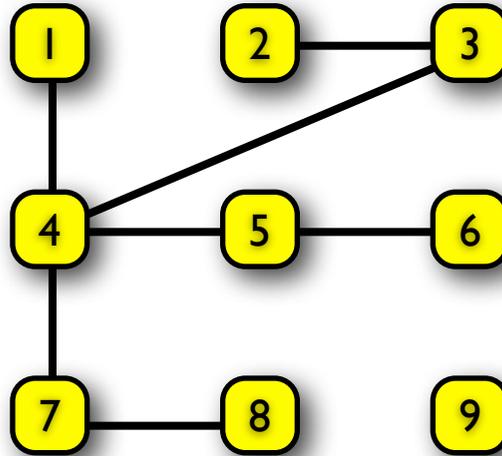
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



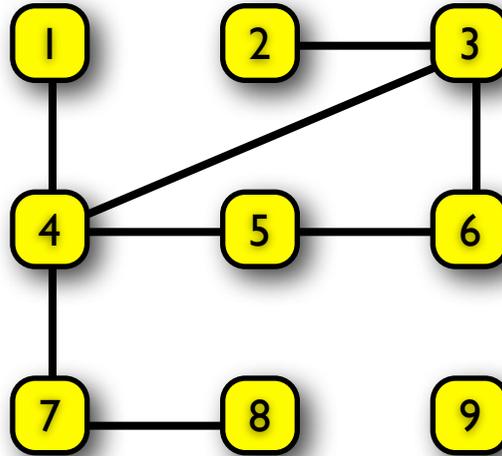
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



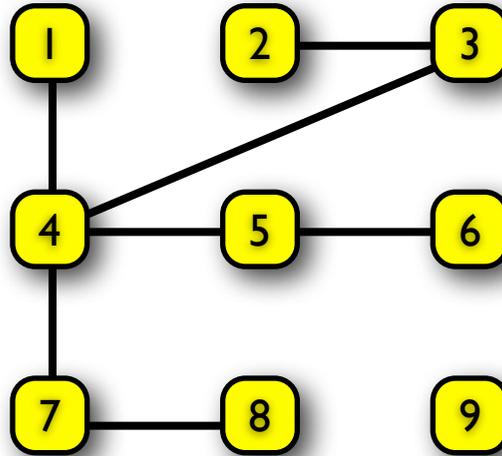
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



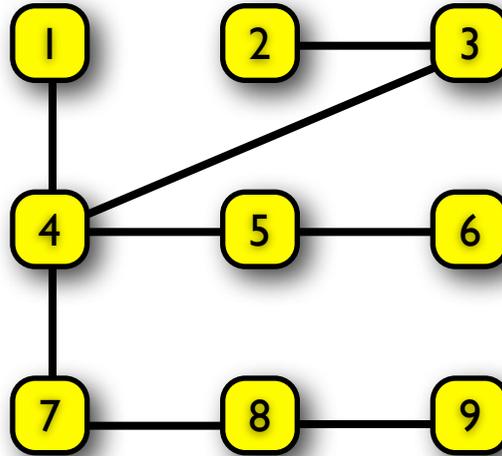
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



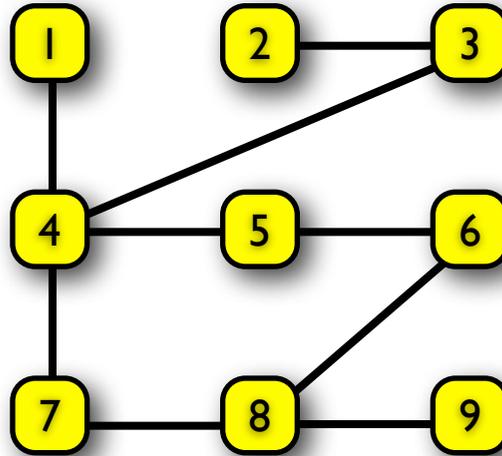
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



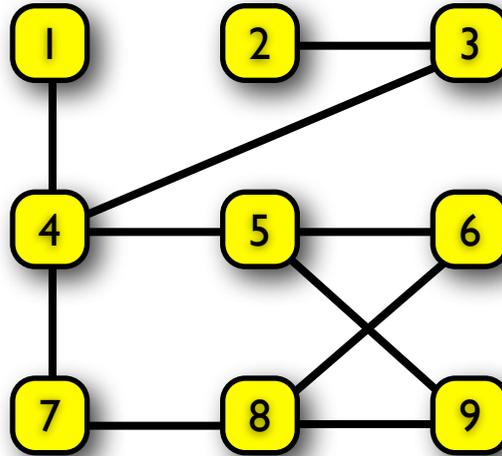
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



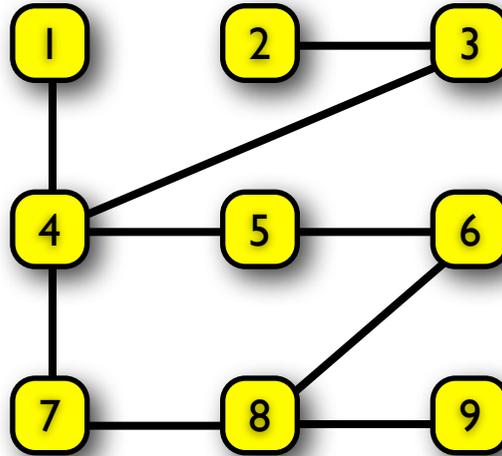
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



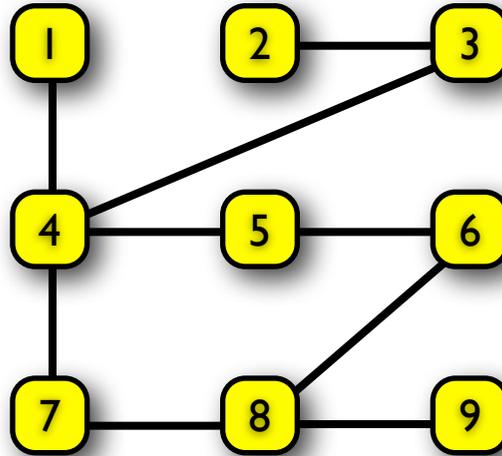
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



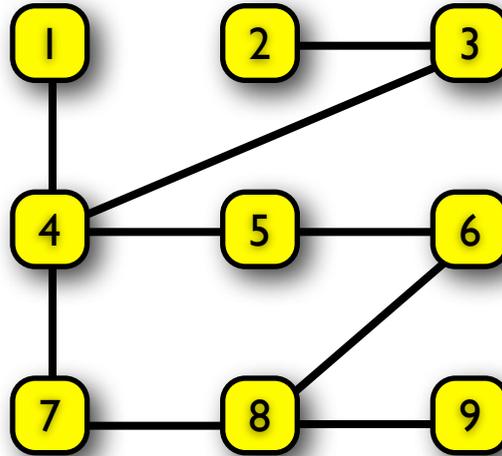
- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.

# Spanners Algorithm



- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.
- **Lemma:** All distances preserved up to a factor  $2t-1$  because an edge  $(u,v)$  was only ignored if there was already a path of length at most  $2t-1$  between  $u$  and  $v$ .

# Spanners Algorithm



- **Algorithm:** Store next edge  $(u,v)$  unless it completes a cycle of length  $2t$  or less.
- **Lemma:** All distances preserved up to a factor  $2t-1$  because an edge  $(u,v)$  was only ignored if there was already a path of length at most  $2t-1$  between  $u$  and  $v$ .
- **Lemma:** At most  $(n^{1+1/t})$  edges stored since shortest cycle among stored edges has length at least  $2t+1$ .

# Other Algorithms

# Other Algorithms

- Matchings: See Sudipto's talk...
  - ▶ **Goal**: Find large set of disjoint edges.
  - ▶ **Results**:  $\tilde{O}(n)$ -space algorithms 2-approx. (unweighted) and 4-approx. (weighted). Can do better if edges are grouped together by end-point or arrive in random order.
  - ▶ **Extensions**:  $O(1)$  approx. for various sub-modular problems.

# Other Algorithms

- Matchings: See Sudipto's talk...
  - ▶ **Goal**: Find large set of disjoint edges.
  - ▶ **Results**:  $\tilde{O}(n)$ -space algorithms 2-approx. (unweighted) and 4-approx. (weighted). Can do better if edges are grouped together by end-point or arrive in random order.
  - ▶ **Extensions**:  $O(1)$  approx. for various sub-modular problems.
- Counting Triangles: Estimate the number of triangles (or small cycle or clique etc.). See Srikanta's talk...

# Other Algorithms

- Matchings: See Sudipto's talk...
  - ▶ **Goal**: Find large set of disjoint edges.
  - ▶ **Results**:  $\tilde{O}(n)$ -space algorithms 2-approx. (unweighted) and 4-approx. (weighted). Can do better if edges are grouped together by end-point or arrive in random order.
  - ▶ **Extensions**:  $O(1)$  approx. for various sub-modular problems.
- Counting Triangles: Estimate the number of triangles (or small cycle or clique etc.). See Srikanta's talk...
- Random Walks: Simulate length  $t$  random walks in  $\sqrt{t}$  passes.

# Other Algorithms

- Matchings: See Sudipto's talk...
  - ▶ **Goal**: Find large set of disjoint edges.
  - ▶ **Results**:  $\tilde{O}(n)$ -space algorithms 2-approx. (unweighted) and 4-approx. (weighted). Can do better if edges are grouped together by end-point or arrive in random order.
  - ▶ **Extensions**:  $O(1)$  approx. for various sub-modular problems.
- Counting Triangles: Estimate the number of triangles (or small cycle or clique etc.). See Srikanta's talk...
- Random Walks: Simulate length  $t$  random walks in  $\sqrt{t}$  passes.
- Other: Minimum spanning trees, bipartiteness, finding dense components, correlation clustering, independent sets, etc.

**1. Algorithms**

**2. Extensions**

**3. Directions**

**1. Algorithms**

**2. Extensions**

**3. Directions**

# Extensions of Model

# Extensions of Model

- Sliding Window: Infinite stream but only consider graph defined by recent  $w$  edges. Can solve most aforementioned problems.

# Extensions of Model

- Sliding Window: Infinite stream but only consider graph defined by recent  $w$  edges. Can solve most aforementioned problems.
- Multiple Passes: What's possible with a small number of stream passes? E.g., can find  $(1+\epsilon)$  approx. matching in  $O(\epsilon^{-1})$  passes.

# Extensions of Model

- Sliding Window: Infinite stream but only consider graph defined by recent  $w$  edges. Can solve most aforementioned problems.
- Multiple Passes: What's possible with a small number of stream passes? E.g., can find  $(1+\epsilon)$  approx. matching in  $O(\epsilon^{-1})$  passes.
- Annotated Streams: Suppose a third party “annotates” the stream to assist with the computation. Can we reduce required memory while still verifying correctness.

# Extensions of Model

- Sliding Window: Infinite stream but only consider graph defined by recent  $w$  edges. Can solve most aforementioned problems.
- Multiple Passes: What's possible with a small number of stream passes? E.g., can find  $(1+\epsilon)$  approx. matching in  $O(\epsilon^{-1})$  passes.
- Annotated Streams: Suppose a third party “annotates” the stream to assist with the computation. Can we reduce required memory while still verifying correctness.

STREAM



# Extensions of Model

- Sliding Window: Infinite stream but only consider graph defined by recent  $w$  edges. Can solve most aforementioned problems.
- Multiple Passes: What's possible with a small number of stream passes? E.g., can find  $(1+\epsilon)$  approx. matching in  $O(\epsilon^{-1})$  passes.
- Annotated Streams: Suppose a third party “annotates” the stream to assist with the computation. Can we reduce required memory while still verifying correctness.



# Dynamic Graphs

# Dynamic Graphs

- Dynamic Graph Streams: Suppose the stream consists of edges both being added and removed from the underlying graph.
- Can we maintain a uniform edge sample in small space?
  - ▶ **Challenge**: The sampled edge we have remembered so far may be deleted at the next step.
  - ▶ **Result**: Can maintain uniform sample in  $O(\text{polylog } n)$  space via a technique called “ $l_0$  sampling”.

# Dynamic Graphs

- Dynamic Graph Streams: Suppose the stream consists of edges both being added and removed from the underlying graph.
- Can we maintain a uniform edge sample in small space?
  - ▶ **Challenge**: The sampled edge we have remembered so far may be deleted at the next step.
  - ▶ **Result**: Can maintain uniform sample in  $O(\text{polylog } n)$  space via a technique called “ $l_0$  sampling”.
- More powerful sampling techniques:
  - ▶ In  $O(n \text{ polylog } n)$  space, can construct a data structure that returns a random edge across any queried cut.
  - ▶ In  $O(n \text{ polylog } n)$  space, can sample edges where  $(u,v)$  is sampled w/p inversely proportional to size of min  $u$ - $v$  cut.

# Distributed Graph Data

# Distributed Graph Data



...

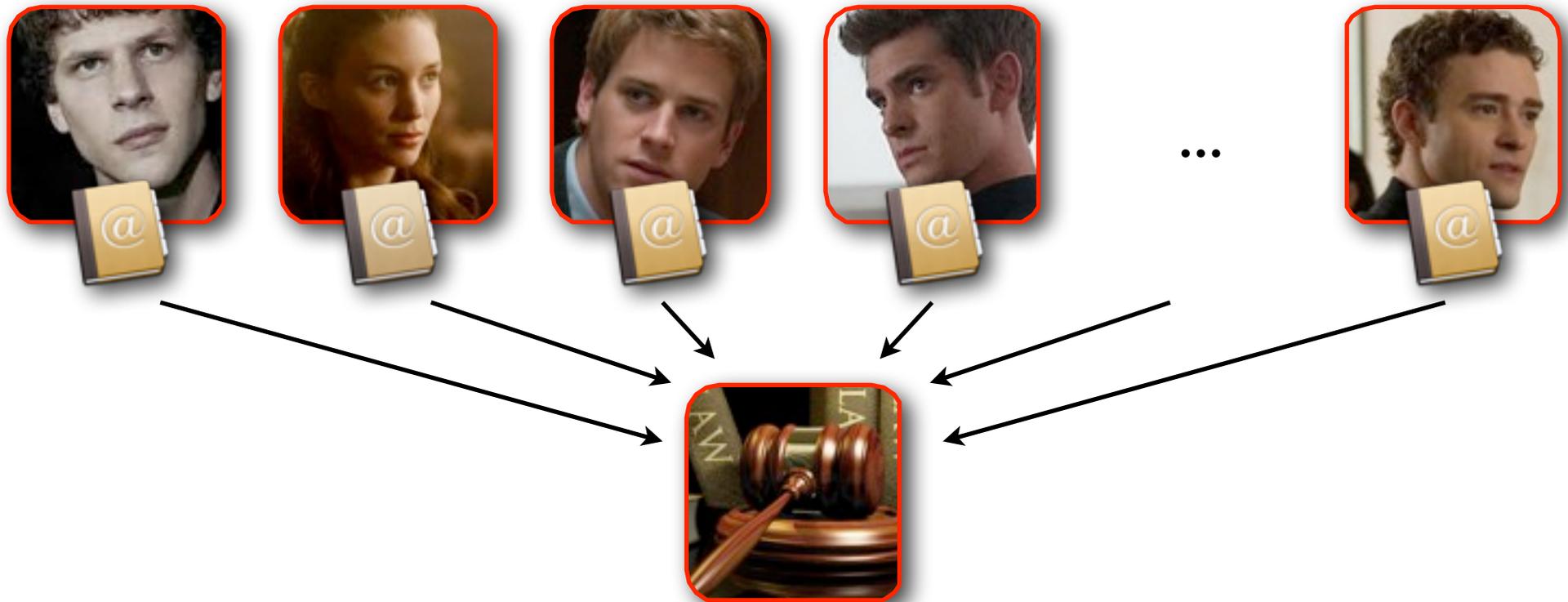


# Distributed Graph Data



- ***Setting:*** The rows of an adjacency matrix are partitioned between different machines. Equivalently, consider  $n$  players each of whom has an “address book” listing their friends.

# Distributed Graph Data



- ***Setting:*** The rows of an adjacency matrix are partitioned between different machines. Equivalently, consider  $n$  players each of whom has an “address book” listing their friends.
- ***Goal:*** Each player sends a “short” message to a third party who then determines if underlying graph is connected.

# Distributed Graph Data



- Appears that some messages need to be  $\Omega(n)$  bits: If there's a **bridge**  $(u,v)$  in the graph, one of the friends needs to mention this friendship but neither friend knows it's a bridge.

# Distributed Graph Data



- Appears that some messages need to be  $\Omega(n)$  bits: If there's a **bridge**  $(u,v)$  in the graph, one of the friends needs to mention this friendship but neither friend knows it's a bridge.
- Thm:  $O(\text{polylog } n)$  bit messages suffice!

# Distributed Graph Data



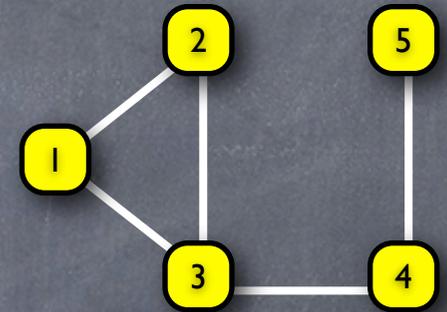
- Appears that some messages need to be  $\Omega(n)$  bits: If there's a **bridge**  $(u,v)$  in the graph, one of the friends needs to mention this friendship but neither friend knows it's a bridge.
- Thm:  $O(\text{polylog } n)$  bit messages suffice!
  - ▶ With a small increase of size, can allow third-party to estimate all cut sizes and spectral properties of the graph!
  - ▶ Protocol is based on "**cut sampling**" primitive where third party can deduce some edge across any cut w/p  $1 - 1/\text{poly}(n)$

# Basic Idea: Cut Sampling

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $a_i$ .

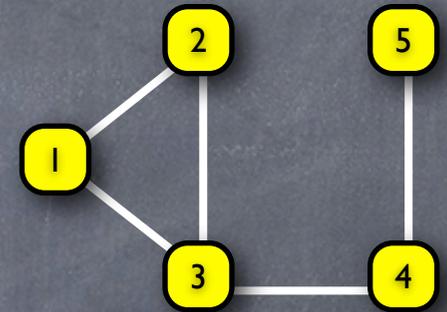
$$\mathbf{a}_1 = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

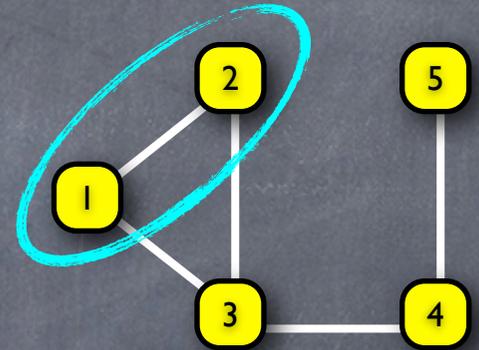
$$\begin{array}{l} \mathbf{a}_1 = \left( \begin{array}{cccccccccc} \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right) \\ \mathbf{a}_2 = \left( \begin{array}{cccccccccc} -1 & 0 & 0 & 0 & 1 & 0 &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



# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



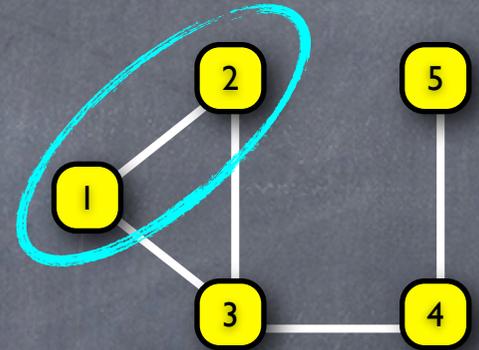
# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

$$\mathbf{a}_1 = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

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



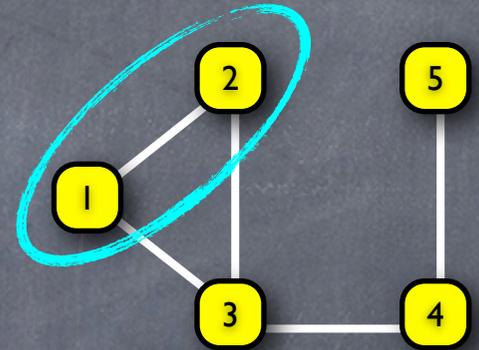
# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

$$\mathbf{a}_1 = \begin{pmatrix} \{1,2\} & \{1,3\} & \{1,4\} & \{1,5\} & \{2,3\} & \{2,4\} & \{2,5\} & \{3,4\} & \{3,5\} & \{4,5\} \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

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

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



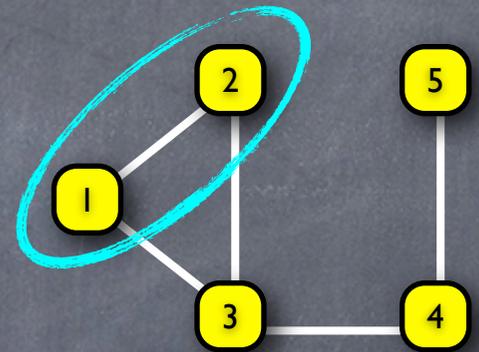
- **Lemma:** For any subset of nodes  $S \subset V$ ,

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

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



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

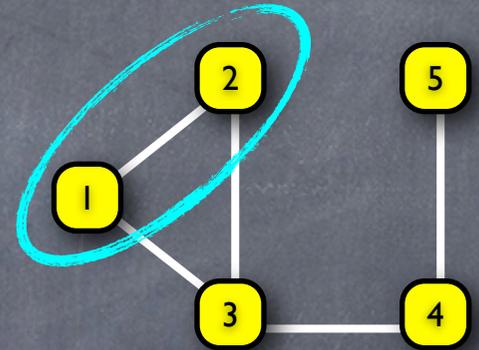
$$\text{support} \left( \sum_{i \in S} \mathbf{a}_i \right) = E(S, V \setminus S)$$

- **Second:** Send  $M\mathbf{a}_j$  where  $M$  is random projection to  $\mathbb{R}^{\text{polylog } N}$  such that for any  $\mathbf{a}$ , we can infer non-zero entry of  $\mathbf{a}$  from  $M\mathbf{a}$ . [Jowhari, Saglam, Tardos 2011]

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



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

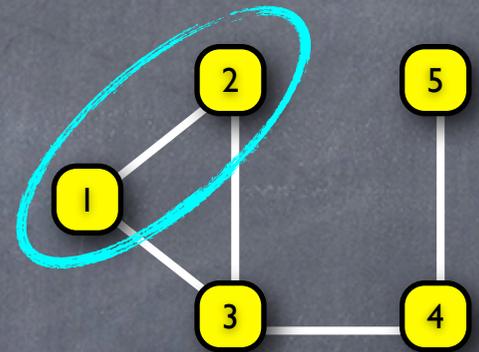
$$\text{support} \left( \sum_{i \in S} \mathbf{a}_i \right) = E(S, V \setminus S)$$

- **Second:** Send  $M\mathbf{a}_j$  where  $M$  is random projection to  $\mathbb{R}^{\text{polylog } N}$  such that for any  $\mathbf{a}$ , we can infer non-zero entry of  $\mathbf{a}$  from  $M\mathbf{a}$ . [Jowhari, Saglam, Tardos 2011]
- To get incident edge on component  $S \subset V$  use:

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



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

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

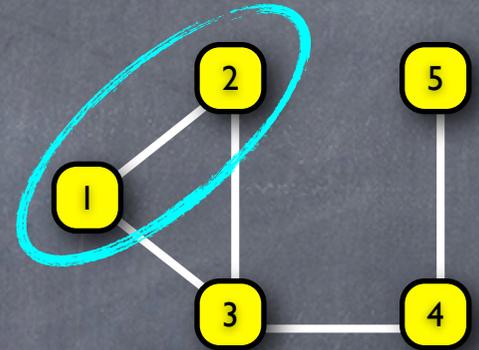
- **Second:** Send  $M\mathbf{a}_j$  where  $M$  is random projection to  $\mathbb{R}^{\text{polylog } N}$  such that for any  $\mathbf{a}$ , we can infer non-zero entry of  $\mathbf{a}$  from  $M\mathbf{a}$ . [Jowhari, Saglam, Tardos 2011]
- To get incident edge on component  $S \subset V$  use:

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

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



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

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

- **Second:** Send  $M\mathbf{a}_j$  where  $M$  is random projection to  $\mathbb{R}^{\text{polylog } N}$  such that for any  $\mathbf{a}$ , we can infer non-zero entry of  $\mathbf{a}$  from  $M\mathbf{a}$ . [Jowhari, Saglam, Tardos 2011]

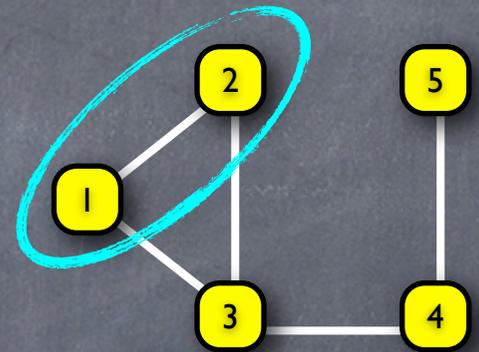
- To get incident edge on component  $S \subset V$  use:

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

# Basic Idea: Cut Sampling

- **First:** Encode neighborhoods of  $i$  as vector  $\mathbf{a}_i$ .

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



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

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

- **Second:** Send  $M\mathbf{a}_j$  where  $M$  is random projection to  $\mathbb{R}^{\text{polylog } N}$  such that for any  $\mathbf{a}$ , we can infer non-zero entry of  $\mathbf{a}$  from  $M\mathbf{a}$ . [Jowhari, Saglam, Tardos 2011]

- To get incident edge on component  $S \subset V$  use:

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

**1. Algorithms**

**2. Extensions**

**3. Directions**

**1. Algorithms**

**2. Extensions**

**3. Directions**

# Open Problems

# Open Problems

? Many specific open questions:

- Approximate matching in  $\tilde{O}(n)$ -space with deletions?
- Can we construct spanners of sliding window graphs?
- Improve approx. factors for matchings and triangles...

# Open Problems

? Many specific open questions:

- Approximate matching in  $\tilde{O}(n)$ -space with deletions?
- Can we construct spanners of sliding window graphs?
- Improve approx. factors for matchings and triangles...

? Open Problems Wiki: Large set of open problems in data streams and property testing can be found at:

<http://sublinear.info>

# Future Directions

# Future Directions

- ? Directed Graphs: Almost all research to date has considered undirected graphs but many natural graphs are directed. May need multiple passes but  $O(\log n)$  passes might be sufficient.

# Future Directions

- ? Directed Graphs: Almost all research to date has considered undirected graphs but many natural graphs are directed. May need multiple passes but  $O(\log n)$  passes might be sufficient.
- ? Stream Ordering: Consider problems under different orderings, e.g., grouped-by-endpoint, increasing weight, random order.

# Future Directions

- ? Directed Graphs: Almost all research to date has considered undirected graphs but many natural graphs are directed. May need multiple passes but  $O(\log n)$  passes might be sufficient.
- ? Stream Ordering: Consider problems under different orderings, e.g., grouped-by-endpoint, increasing weight, random order.
- ? More or Less Space: Most work has focus on  $\tilde{O}(n)$ -space algorithms. Can we reduce space-complexity for specific families of graphs? What's possible with slightly more space?

# Future Directions

- ? Directed Graphs: Almost all research to date has considered undirected graphs but many natural graphs are directed. May need multiple passes but  $O(\log n)$  passes might be sufficient.
- ? Stream Ordering: Consider problems under different orderings, e.g., grouped-by-endpoint, increasing weight, random order.
- ? More or Less Space: Most work has focus on  $\tilde{O}(n)$ -space algorithms. Can we reduce space-complexity for specific families of graphs? What's possible with slightly more space?
- ? Explore deeper connections with distributed algorithms, communication complexity, dynamic graphs data structures...

# Summary of the Survey



*Thanks!*

# Summary of the Survey

- Algorithms: Spanners and sparsifiers capture different properties of the graph. Efficient constructions in streaming model. Other positive results for matchings, triangles, etc.
- Extensions: Many variants of the basic model including sliding windows, multi-pass, edge deletions, annotations...
- Directions: Improve existing results. Future directions include directed graphs, stream ordering, specific graph families etc.



*Thanks!*



# Lower Bound for Connectivity

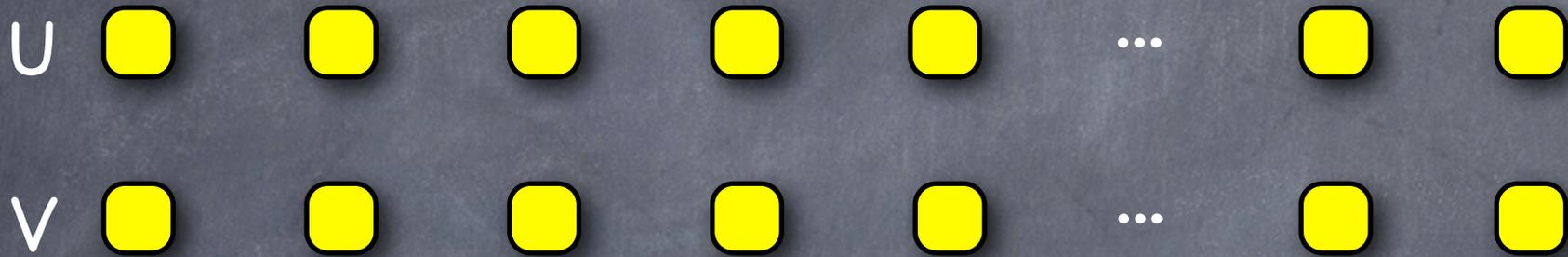
# Lower Bound for Connectivity

- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.

# Lower Bound for Connectivity

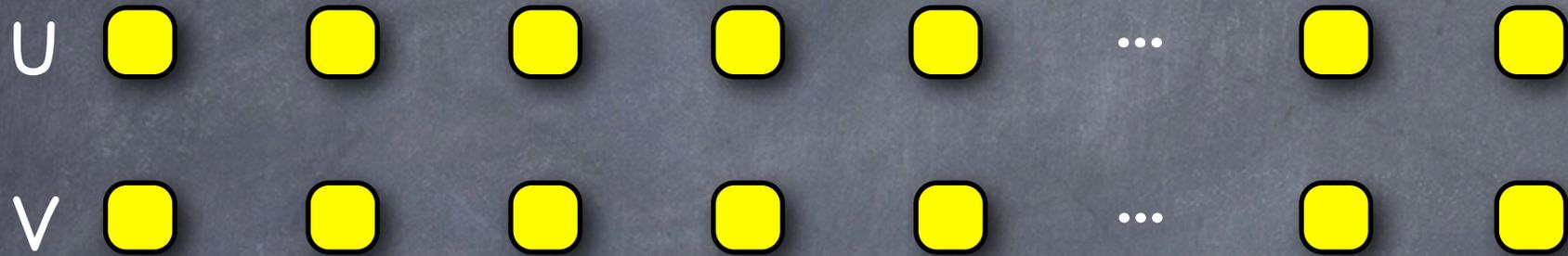
- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.

# Lower Bound for Connectivity



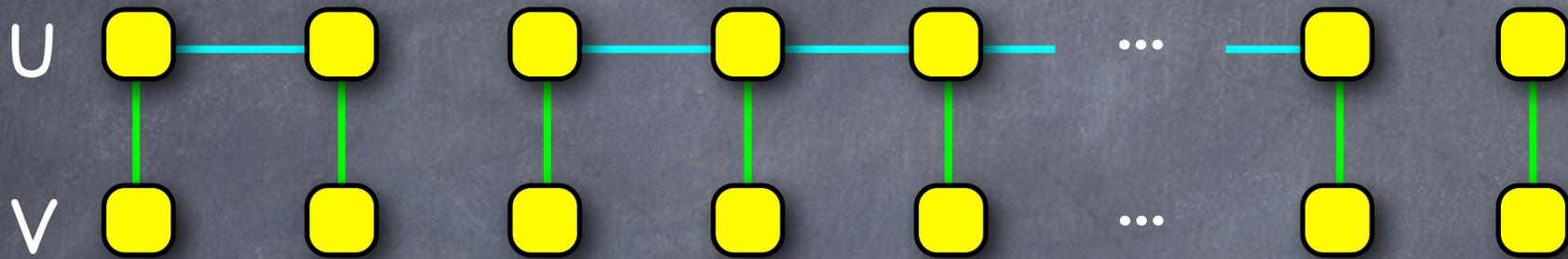
- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$

# Lower Bound for Connectivity



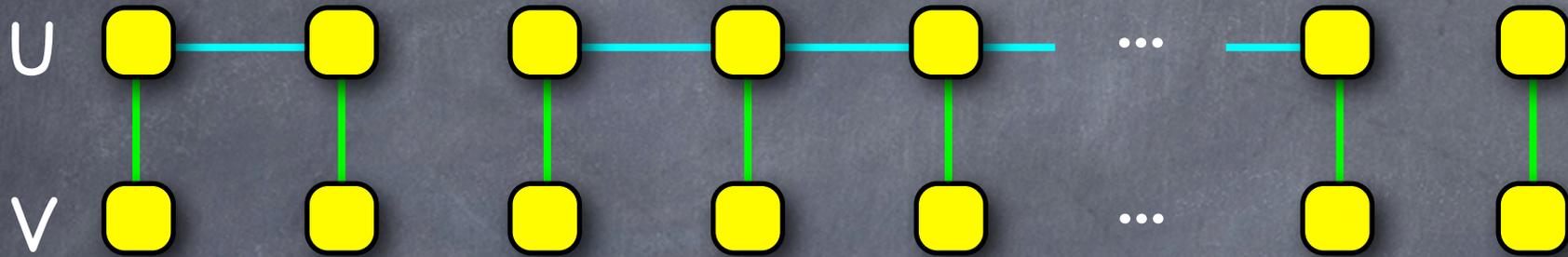
- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$
- Alice runs  $A$  on  $E_1 = \{u_i v_i : 1 \leq i \leq n\}$  and  $E_2 = \{u_i u_{i+1} : x_i = 0\}$

# Lower Bound for Connectivity



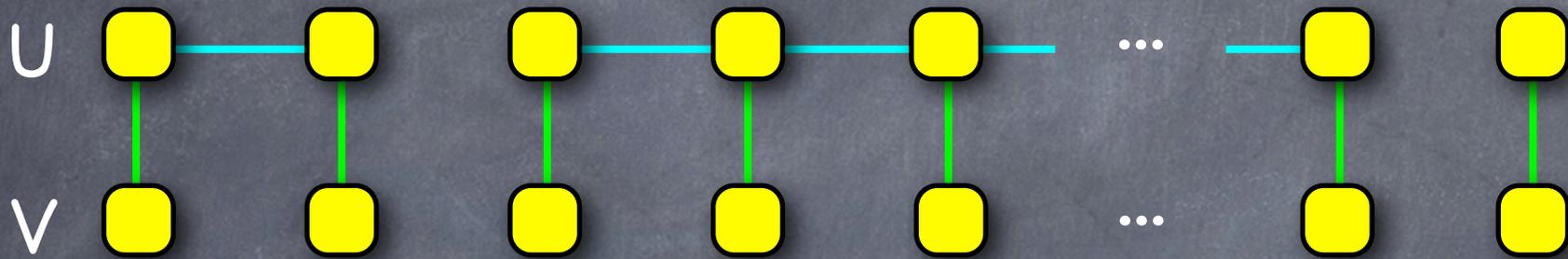
- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$
- Alice runs  $A$  on  $E_1 = \{u_i v_i : 1 \leq i \leq n\}$  and  $E_2 = \{u_i u_{i+1} : x_i = 0\}$

# Lower Bound for Connectivity



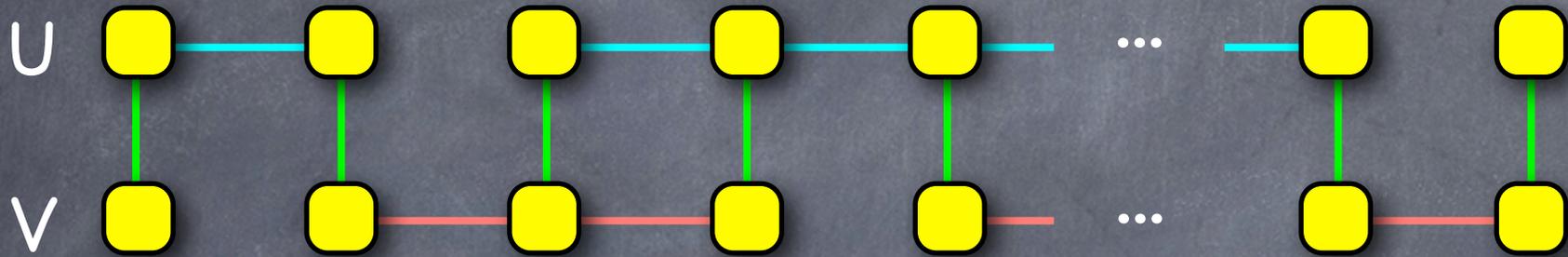
- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$
- Alice runs  $A$  on  $E_1 = \{u_i v_i : 1 \leq i \leq n\}$  and  $E_2 = \{u_i u_{i+1} : x_i = 0\}$
- Send memory to Bob who runs  $A$  on  $E_3 = \{v_i v_{i+1} : y_i = 0\}$

# Lower Bound for Connectivity



- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$
- Alice runs  $A$  on  $E_1 = \{u_i v_i : 1 \leq i \leq n\}$  and  $E_2 = \{u_i u_{i+1} : x_i = 0\}$
- Send memory to Bob who runs  $A$  on  $E_3 = \{v_i v_{i+1} : y_i = 0\}$
- Output of  $A$  resolves matrix question so  $s = \Omega(n)$ .

# Lower Bound for Connectivity



- Alice and Bob have  $x, y \in \{0, 1\}^n$ . For Bob to check if  $x_i = y_i = 1$  for some  $i$  needs  $\Omega(n)$  communication.
- Let  $A$  be an  $s$  space algorithm for connectivity.
- Consider 2-layer graph  $(U, V)$  with  $|U| = |V| = n$
- Alice runs  $A$  on  $E_1 = \{u_i v_i : 1 \leq i \leq n\}$  and  $E_2 = \{u_i u_{i+1} : x_i = 0\}$
- Send memory to Bob who runs  $A$  on  $E_3 = \{v_i v_{i+1} : y_i = 0\}$
- Output of  $A$  resolves matrix question so  $s = \Omega(n)$ .