Data Streams & Communication Complexity

Lecture 2: Graph Spanners, Sparsifiers, & Sketches

Andrew McGregor, UMass Amherst

Graph Streams

- Consider a stream of m edges

$$\langle e_1, e_2, \ldots \quad \ldots, e_m \rangle$$

defining a graph G with nodes $V = [n]$ and $E = \{e_1, \ldots, e_m\}$

Graph Streams

- Consider a stream of m edges

$$\langle e_1, e_2, \ldots \quad \ldots, e_m \rangle$$

  defining a graph G with nodes $V = [n]$ and $E = \{e_1, \ldots, e_m\}$
- *Semi-streaming:* What can we compute with $O(n \cdot \operatorname{polylog} n)$ space?

Outline

Spanners and Distances

Sparsifiers and Cuts

Sketches and Dynamic Graphs

- Connectivity
- k-Connectivity
- Minimum Cut

Outline

Graph Distances

- *Goal:* Approximate length of the shortest path $d_G(u, v)$ between a pair of nodes $u, v \in G$,

Graph Distances

- *Goal:* Approximate length of the shortest path $d_G(u, v)$ between a pair of nodes $u, v \in G$,

Definition

An α-spanner of graph G is a subgraph H such that for any nodes u, v,

$$d_G(u, v) \leq d_H(u, v) \leq \alpha d_G(u, v) \ .$$

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$
  - For each edge (u, v), if u and v aren't connected in F,

$$F \leftarrow F \cup \{(u, v)\}$$

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$
  - For each edge (u, v), if u and v aren't connected in F,

$$F \leftarrow F \cup \{(u, v)\}$$

- *Analysis:*

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$
  - For each edge (u, v), if u and v aren't connected in F,

$$F \leftarrow F \cup \{(u, v)\}$$

- *Analysis:*
  - F has the same number of connected components as G

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$
  - For each edge (u, v), if u and v aren't connected in F,

$$F \leftarrow F \cup \{(u, v)\}$$

- *Analysis:*
  - F has the same number of connected components as G
  - F has at most $n - 1$ edges.

Warm-Up: Connectivity

- *Goal:* Compute the number of connected components.
- *Algorithm:* Maintain a spanning forest F
  - $F \leftarrow \emptyset$
  - For each edge (u, v), if u and v aren't connected in F,
    $$F \leftarrow F \cup \{(u, v)\}$$
- *Analysis:*
  - F has the same number of connected components as G
  - F has at most $n - 1$ edges.
- *Thm:* Can count connected components in $O(n \log n)$ space.

Spanners

- *Algorithm:*

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.
  - For each edge (u, v), if $d_H(u, v) \geq 2t$, $H \leftarrow H \cup \{(u, v)\}$

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.
  - For each edge (u, v), if $d_H(u, v) \geq 2t$, $H \leftarrow H \cup \{(u, v)\}$
- *Analysis:*

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.
  - For each edge (u, v), if $d_H(u, v) \geq 2t$, $H \leftarrow H \cup \{(u, v)\}$
- *Analysis:*
  - Distances increase by at most a factor $2t - 1$ since an edge (u, v) is only forgotten if there's already a detour of length at most $2t - 1$.

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.
  - For each edge (u, v), if $d_H(u, v) \geq 2t$, $H \leftarrow H \cup \{(u, v)\}$
- *Analysis:*
  - Distances increase by at most a factor $2t - 1$ since an edge (u, v) is only forgotten if there's already a detour of length at most $2t - 1$.
  - *Lemma:* H has $O(n^{1+1/t})$ edges since all cycles have length $\geq 2t + 1$.

Spanners

- *Algorithm:*
  - $H \leftarrow \emptyset$.
  - For each edge (u, v), if $d_H(u, v) \geq 2t$, $H \leftarrow H \cup \{(u, v)\}$
- *Analysis:*
  - Distances increase by at most a factor $2t - 1$ since an edge (u, v) is only forgotten if there's already a detour of length at most $2t - 1$.
  - *Lemma:* H has $O(n^{1+1/t})$ edges since all cycles have length $\geq 2t + 1$.

Theorem

Can $(2t - 1)$-approximate all distances using only $O(n^{1+1/t})$ space.

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

Proof of Lemma

Lemma
A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.
- Let J be the graph formed by removing all nodes with degree less than $d/2$.

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.
- Let J be the graph formed by removing all nodes with degree less than $d/2$. Note $J \neq \emptyset$ because $< n(d/2) = m$ edges are removed.

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.
- Let J be the graph formed by removing all nodes with degree less than $d/2$. Note $J \neq \emptyset$ because $< n(d/2) = m$ edges are removed.
- Grow a BFS of depth t from an arbitrary node in J.

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.
- Let J be the graph formed by removing all nodes with degree less than $d/2$. Note $J \neq \emptyset$ because $< n(d/2) = m$ edges are removed.
- Grow a BFS of depth t from an arbitrary node in J.
- Because a) no cycles of length less than $2t + 1$ and b) all degrees in J are at least $d/2$, number of nodes at t-th level of BFS is at least

$$(d/2 - 1)^t = (m/n - 1)^t$$

Proof of Lemma

Lemma

A graph H on n nodes with no cycles of length $\leq 2t$ has $O(n^{1+1/t})$ edges.

- Let $d = 2m/n$ be average degree of H.
- Let J be the graph formed by removing all nodes with degree less than $d/2$. Note $J \neq \emptyset$ because $< n(d/2) = m$ edges are removed.
- Grow a BFS of depth t from an arbitrary node in J.
- Because a) no cycles of length less than $2t + 1$ and b) all degrees in J are at least $d/2$, number of nodes at t-th level of BFS is at least

$$(d/2 - 1)^t = (m/n - 1)^t$$

- But $(m/n - 1)^t \leq |J| \leq n$ and therefore $m \leq n + n^{1+1/t}$.

Outline

Cuts and Sparsifiers

- *Goal:* Approximate capacity $C_G(S)$ of any cut $(S, V \setminus S)$ in G.

Cuts and Sparsifiers

- *Goal:* Approximate capacity $C_G(S)$ of any cut $(S, V \setminus S)$ in G.

Definition

An α-sparsifier of graph G is a weighted subgraph H such that for any cut $(S, V \setminus S)$,

$$C_G(S) \leq C_H(S) \leq \alpha C_G(S) \ .$$

where C_G and C_H is the capacity of the cut in G and H respectively.

Cuts and Sparsifiers

- *Goal:* Approximate capacity $C_G(S)$ of any cut $(S, V \setminus S)$ in G.

Definition

An α-sparsifier of graph G is a weighted subgraph H such that for any cut $(S, V \setminus S)$,

$$C_G(S) \leq C_H(S) \leq \alpha C_G(S) \ .$$

where C_G and C_H is the capacity of the cut in G and H respectively.

Theorem (Batson, Spielman, Srivastava)

Exists offline algorithm $\mathcal{A}$ returning $(1+\epsilon)$-sparsifier with $O(n\epsilon^{-2})$ edges.

Cuts and Sparsifiers

- *Goal:* Approximate capacity $C_G(S)$ of any cut $(S, V \setminus S)$ in G.

Definition

An α-sparsifier of graph G is a weighted subgraph H such that for any cut $(S, V \setminus S)$,

$$C_G(S) \leq C_H(S) \leq \alpha C_G(S) \ .$$

where C_G and C_H is the capacity of the cut in G and H respectively.

Theorem (Batson, Spielman, Srivastava)

Exists offline algorithm $\mathcal{A}$ returning $(1+\epsilon)$-sparsifier with $O(n\epsilon^{-2})$ edges.

- *Idea:* Use $\mathcal{A}$ as a black box to recursively sparsify graph stream.

Basic Properties of Sparsifiers

Lemma

If H_1 and H_2 are α-sparsifiers of G_1 and G_2. Then $H_1 \cup H_2$ is an α-sparsifier of $G_1 \cup G_2$.

Basic Properties of Sparsifiers

Lemma

If H_1 and H_2 are α-sparsifiers of G_1 and G_2. Then $H_1 \cup H_2$ is an α-sparsifier of $G_1 \cup G_2$.

Lemma

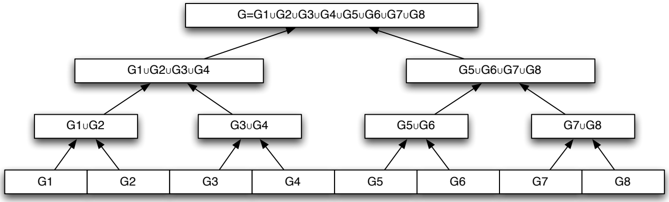
If J is an α-sparsifiers of H and H is an α-sparsifier of G. Then J is an α^2-sparsifier of G.

Stream Sparsification

- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.

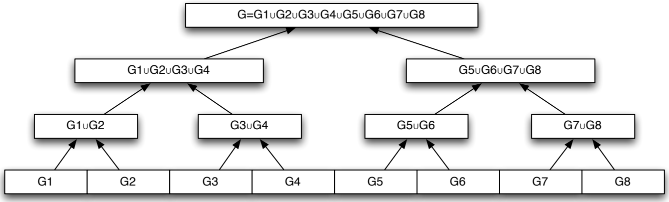
Stream Sparsification

- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



Stream Sparsification

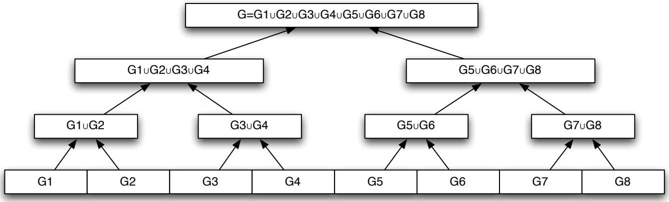
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1 + \gamma$:

Stream Sparsification

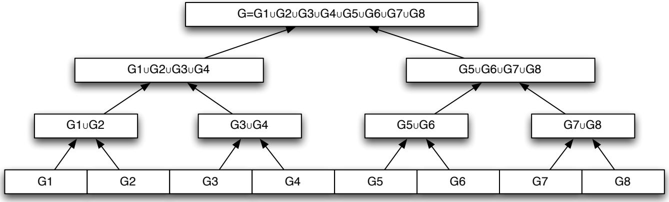
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1

Stream Sparsification

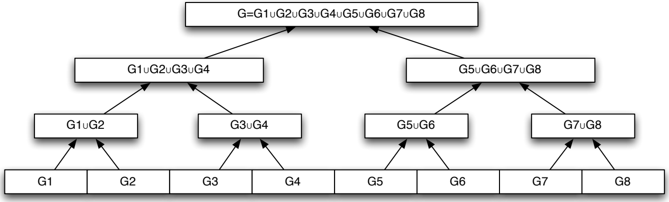
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1 + \gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2

Stream Sparsification

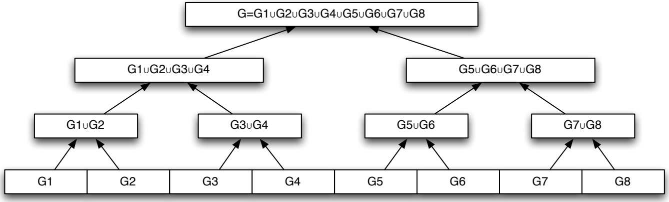
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$

Stream Sparsification

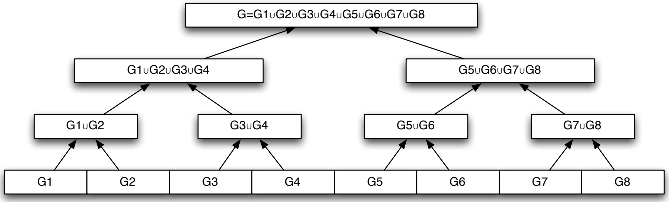
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3

Stream Sparsification

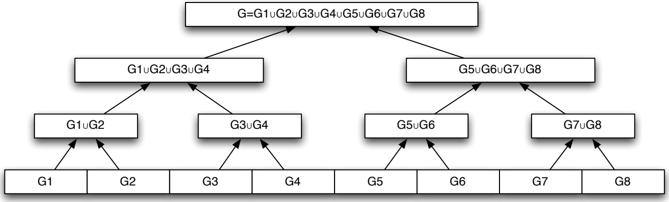
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3
  - Read in G_4: compute $\mathcal{A}(G_4)$ and forget G_4

Stream Sparsification

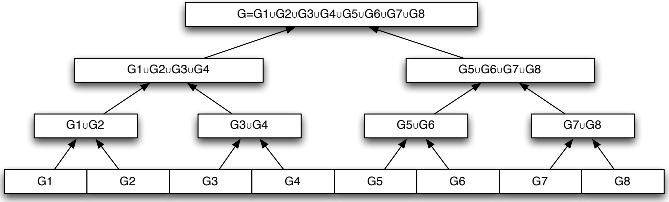
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3
  - Read in G_4: compute $\mathcal{A}(G_4)$ and forget G_4
  - Compute $\mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4))$ and forget $\mathcal{A}(G_3)$ and $\mathcal{A}(G_4)$

Stream Sparsification

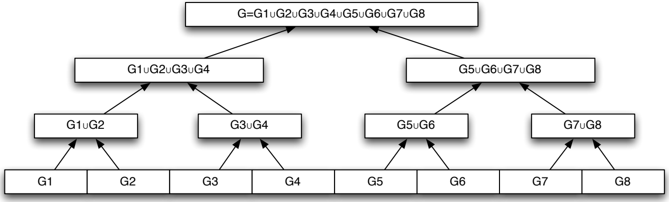
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3
  - Read in G_4: compute $\mathcal{A}(G_4)$ and forget G_4
  - Compute $\mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4))$ and forget $\mathcal{A}(G_3)$ and $\mathcal{A}(G_4)$
  - Compute $\mathcal{A}(\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2)) \cup \mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4)))$ and forget ...

Stream Sparsification

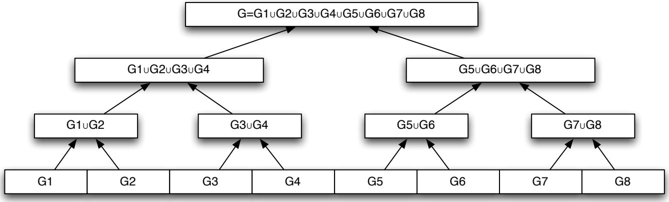
- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1+\gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3
  - Read in G_4: compute $\mathcal{A}(G_4)$ and forget G_4
  - Compute $\mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4))$ and forget $\mathcal{A}(G_3)$ and $\mathcal{A}(G_4)$
  - Compute $\mathcal{A}(\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2)) \cup \mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4)))$ and forget ...
- Results in a $(1+\gamma)^{\log m}$-sparsifier for G in $O(n\gamma^{-2} \log m)$ space.

Stream Sparsification

- Divide stream into segments $G_1, G_2, \ldots$ each of $t = O(n\epsilon^{-2})$ edges.
- Consider binary tree over segments



- Recursively use $\mathcal{A}$ with parameter $1 + \gamma$:
  - Read in G_1: compute $\mathcal{A}(G_1)$ and forget G_1
  - Read in G_2: compute $\mathcal{A}(G_2)$ and forget G_2
  - Compute $\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2))$ and forget $\mathcal{A}(G_1)$ and $\mathcal{A}(G_2)$
  - Read in G_3: compute $\mathcal{A}(G_3)$ and forget G_3
  - Read in G_4: compute $\mathcal{A}(G_4)$ and forget G_4
  - Compute $\mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4))$ and forget $\mathcal{A}(G_3)$ and $\mathcal{A}(G_4)$
  - Compute $\mathcal{A}(\mathcal{A}(\mathcal{A}(G_1) \cup \mathcal{A}(G_2)) \cup \mathcal{A}(\mathcal{A}(G_3) \cup \mathcal{A}(G_4)))$ and forget ...
- Results in a $(1 + \gamma)^{\log m}$-sparsifier for G in $O(n\gamma^{-2} \log m)$ space.
- If $\gamma = O(\epsilon / \log m)$, we get $(1 + \epsilon)$-sparsifier in $O(n\epsilon^{-2} \log^3 m)$ space.

Outline

Spanners and Distances

Sparsifiers and Cuts

Sketches and Dynamic Graphs

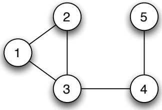
- Connectivity
- k-Connectivity
- Minimum Cut

Dynamic Graph Streams

- Consider a stream of edges inserts and deletions, e.g.,

  $\langle \texttt{add}(1,2), \texttt{add}(1,4), \texttt{add}(2,3), \texttt{add}(1,3), \texttt{add}(4,5), \texttt{add}(3,4), \texttt{del}(1,4) \rangle$

  would result in the following graph

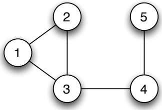


Dynamic Graph Streams

- Consider a stream of edges inserts and deletions, e.g.,

  $\langle \texttt{add}(1,2), \texttt{add}(1,4), \texttt{add}(2,3), \texttt{add}(1,3), \texttt{add}(4,5), \texttt{add}(3,4), \texttt{del}(1,4) \rangle$

  would result in the following graph



- *Dynamic semi-streaming:* What can we compute about a dynamic graph with only $O(n \cdot \text{polylog}\, n)$ space?

Outline

Connectivity

- *Goal:* Test whether G is connected.

Connectivity

- *Goal:* Test whether G is connected.
- Our algorithm will actually return a spanning forest of G.

Connectivity

- *Goal:* Test whether G is connected.
- Our algorithm will actually return a spanning forest of G.

Lemma

Consider the offline algorithm:

1. *For each node, select an incident edge*
2. *Contract selected edges.*
3. *Repeat until no edges remain.*

After $\log n$ *steps, number of nodes is number of connected components in* G*. Furthermore, set of selected edges contains a spanning forest.*

Connectivity

- *Goal:* Test whether G is connected.
- Our algorithm will actually return a spanning forest of G.

Lemma

Consider the offline algorithm:

1. *For each node, select an incident edge*
2. *Contract selected edges.*
3. *Repeat until no edges remain.*

After $\log n$ *steps, number of nodes is number of connected components in* G*. Furthermore, set of selected edges contains a spanning forest.*

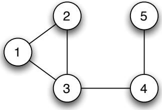
- *Idea:* Emulate above algorithm in a single pass using ℓ_0-sampling of a particular vector representation of G.

Useful Graph Representation

- Represent graph on $[n]$ with edges $E \subset [n] \times [n]$, as matrix
$$G \in \{-1, 0, 1\}^{n \times \binom{n}{2}}$$
with non-zero entries $G_{j,(j,k)} = 1$, $G_{k,(j,k)} = -1$ if $(j, k) \in E$.

Useful Graph Representation

- Represent graph on $[n]$ with edges $E \subset [n] \times [n]$, as matrix
$$G \in \{-1, 0, 1\}^{n \times \binom{n}{2}}$$
with non-zero entries $G_{j,(j,k)} = 1$, $G_{k,(j,k)} = -1$ if $(j,k) \in E$. E.g.,



becomes,

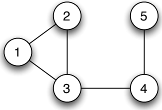
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} (1,2) & (1,3) & (1,4) & (1,5) & (2,3) & (2,4) & (2,5) & (3,4) & (3,5) & (4,5) \end{array} \\
\begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{pmatrix}
\end{array}$$

Useful Graph Representation

- Represent graph on $[n]$ with edges $E \subset [n] \times [n]$, as matrix

$$G \in \{-1, 0, 1\}^{n \times \binom{n}{2}}$$

with non-zero entries $G_{j,(j,k)} = 1$, $G_{k,(j,k)} = -1$ if $(j,k) \in E$. E.g.,



becomes,

$$\begin{array}{c c} & \begin{array}{cccccccccc} (1,2) & (1,3) & (1,4) & (1,5) & (2,3) & (2,4) & (2,5) & (3,4) & (3,5) & (4,5) \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \end{array}$$

- *Lemma:* For $S \subset [n]$, support$(\sum_{i \in S} a_i) = E(S)$ where a_i is ith row of A and $E(S)$ are edges across cut $(S, V \setminus S)$.

Connectivity Algorithm

- Let $A(a_1), A(a_2), \ldots, A(a_n)$ be sketches for ℓ_0 sampling. Can post-process each sketch to find incident edge on each node.

Connectivity Algorithm

- Let $A(a_1), A(a_2), \ldots, A(a_n)$ be sketches for ℓ_0 sampling. Can post-process each sketch to find incident edge on each node.
- Suppose we found edges that connected, e.g., $S = \{a_1, a_2, a_3\}$. How can find an edge $e \in E(S)$ without taking another pass?

Connectivity Algorithm

- Let $A(a_1), A(a_2), \ldots, A(a_n)$ be sketches for ℓ_0 sampling. Can post-process each sketch to find incident edge on each node.
- Suppose we found edges that connected, e.g., $S = \{a_1, a_2, a_3\}$. How can find an edge $e \in E(S)$ without taking another pass?
- *Linearity:* Because of linearity we can just add sketches,

$$A(a_1) + A(a_2) + A(a_3) = A(a_1 + a_2 + a_3) \longrightarrow e \in E(S)$$

Connectivity Algorithm

- Let $A(a_1), A(a_2), \ldots, A(a_n)$ be sketches for ℓ_0 sampling. Can post-process each sketch to find incident edge on each node.
- Suppose we found edges that connected, e.g., $S = \{a_1, a_2, a_3\}$. How can find an edge $e \in E(S)$ without taking another pass?
- *Linearity:* Because of linearity we can just add sketches,

$$A(a_1) + A(a_2) + A(a_3) = A(a_1 + a_2 + a_3) \longrightarrow e \in E(S)$$

- *Under-the-rug:* Actually we need to use $\log n$ independent sketch matrices $B, C, D, \ldots$ to emulate each round of algorithm. But this is fine: we can compute each $B(a_i), C(a_i), D(a_i), \ldots$ during same pass.

Outline

Spanners and Distances

Sparsifiers and Cuts

Sketches and Dynamic Graphs

- Connectivity
- k-Connectivity
- Minimum Cut

k-Connectivity

- *Goal:* Test whether all cuts of G have size at least k.

k-Connectivity

- *Goal:* Test whether all cuts of G have size at least k.
- Our algorithm actually returns a *certificate* of k-connectivity.

k-Connectivity

- *Goal:* Test whether all cuts of G have size at least k.
- Our algorithm actually returns a *certificate* of k-connectivity.

Definition

We say subgraph H is a k-certificate for G if,

$$\forall \text{ cuts } (S, V \setminus S) : \quad C_H(S) \geq \min(C_G(S), k) \ .$$

k-Connectivity

- *Goal:* Test whether all cuts of G have size at least k.
- Our algorithm actually returns a *certificate* of k-connectivity.

Definition

We say subgraph H is a k-certificate for G if,

$$\forall \text{ cuts } (S, V \setminus S): \quad C_H(S) \geq \min(C_G(S), k) \ .$$

Lemma

Let F_1 be a spanning forest of G and, for $i \geq 2$, let F_i be a spanning forest of $G \setminus (F_1 \cup \ldots \cup F_{i-1})$. Then $F_1 \cup \ldots \cup F_k$ is a k-certificate for G.

k-Connectivity

- *Goal:* Test whether all cuts of G have size at least k.
- Our algorithm actually returns a *certificate* of k-connectivity.

Definition

We say subgraph H is a k-certificate for G if,

$$\forall \text{ cuts } (S, V \setminus S) : \quad C_H(S) \geq \min(C_G(S), k) \ .$$

Lemma

Let F_1 be a spanning forest of G and, for $i \geq 2$, let F_i be a spanning forest of $G \setminus (F_1 \cup \ldots \cup F_{i-1})$. Then $F_1 \cup \ldots \cup F_k$ is a k-certificate for G.

- *Idea:* Emulate above algorithm in a single pass by exploiting linearity of CONNECTIVITY algorithm.

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent CONNECTIVITY sketches $A(G)$ and $B(G)$ of the graph G.

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent CONNECTIVITY sketches $A(G)$ and $B(G)$ of the graph G.
    1. Construct F_1 from $A(G)$

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent CONNECTIVITY sketches $A(G)$ and $B(G)$ of the graph G.
  1. Construct F_1 from $A(G)$
  2. Construct $B(F_1)$

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent CONNECTIVITY sketches $A(G)$ and $B(G)$ of the graph G.
  1. Construct F_1 from $A(G)$
  2. Construct $B(F_1)$
  3. Then $B(G) - B(F_1) = B(G \setminus F_1)$ can be used to construct F_2.

k-Connectivity Algorithm

- Can find F_1 using the CONNECTIVITY algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent CONNECTIVITY sketches $A(G)$ and $B(G)$ of the graph G.
    1. Construct F_1 from $A(G)$
    2. Construct $B(F_1)$
    3. Then $B(G) - B(F_1) = B(G \setminus F_1)$ can be used to construct F_2.
- Given $A(G), B(G), C(G)$ we would find F_1 and F_2 as above. We then find F_3 from

$$C(G) - C(F_1) - C(F_2) = C(G \setminus F_1 \cup F_2) \ ,$$

k-Connectivity Algorithm

- Can find F_1 using the $\textsc{connectivity}$ algorithm.
- But how can we find F_2 without taking another pass over the data?
- *Linearity:* Suppose we have independent $\textsc{connectivity}$ sketches $A(G)$ and $B(G)$ of the graph G.
  1. Construct F_1 from $A(G)$
  2. Construct $B(F_1)$
  3. Then $B(G) - B(F_1) = B(G \setminus F_1)$ can be used to construct F_2.
- Given $A(G), B(G), C(G)$ we would find F_1 and F_2 as above. We then find F_3 from

$$C(G) - C(F_1) - C(F_2) = C(G \setminus F_1 \cup F_2) \ ,$$

- And so on... resulting algorithm, $\textsc{connectivity}_k$, requires one pass and uses $O(k \cdot n \cdot \operatorname{polylog} n)$ space.

Outline

Estimating Minimum Cut

- *Goal:* Estimate the size of the min-cut up to a $(1+\epsilon)$ factor.

Estimating Minimum Cut

- *Goal:* Estimate the size of the min-cut up to a $(1+\epsilon)$ factor.
- If min-cut size is $O(\epsilon^{-2} \cdot \text{polylog}\, n)$ then $\textsc{connectivity}_k$ algorithm can find exact min-cut exactly in $O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$ space.

Estimating Minimum Cut

- *Goal:* Estimate the size of the min-cut up to a $(1+\epsilon)$ factor.
- If min-cut size is $O(\epsilon^{-2} \cdot \text{polylog}\, n)$ then $\textsc{connectivity}_k$ algorithm can find exact min-cut exactly in $O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$ space.
- What can be done if min-cut is large?

Estimating Minimum Cut

- *Goal:* Estimate the size of the min-cut up to a $(1+\epsilon)$ factor.
- If min-cut size is $O(\epsilon^{-2} \cdot \text{polylog}\, n)$ then $\textsc{Connectivity}_k$ algorithm can find exact min-cut exactly in $O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$ space.
- What can be done if min-cut is large?

Theorem (Karger)

Let $G = (V, E)$ be an unweighted graph with min-cut value λ. If we sample each edge with probability

$$p \geq p^* := 6\lambda^{-1}\epsilon^{-2} \log n$$

and assign weight $1/p$ to sampled edges, then the resulting graph is an $(1+\epsilon)$-sparsification of G with high probability.

Estimating Minimum Cut

- *Goal:* Estimate the size of the min-cut up to a $(1+\epsilon)$ factor.
- If min-cut size is $O(\epsilon^{-2} \cdot \text{polylog}\, n)$ then $\textsc{connectivity}_k$ algorithm can find exact min-cut exactly in $O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$ space.
- What can be done if min-cut is large?

Theorem (Karger)

Let $G = (V, E)$ be an unweighted graph with min-cut value λ. If we sample each edge with probability

$$p \geq p^* := 6\lambda^{-1}\epsilon^{-2} \log n$$

and assign weight $1/p$ to sampled edges, then the resulting graph is an $(1+\epsilon)$-sparsification of G with high probability.

- *Idea:* Subsample the input graph at different rates and use $\textsc{connectivity}_k$ to compute min-cut size if it's small enough.

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}\left[h_i(e) = 1\right] = 1/2^i$$

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}\left[h_i(e) = 1\right] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}\left[h_i(e) = 1\right] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{Connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}\left[h_i(e) = 1\right] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{Connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

- *Analysis:*
  - Let λ_i be the size of min-cut of G_i

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

- *Analysis:*
  - Let λ_i be the size of min-cut of G_i
  - Karger's result implies $2^i \lambda_i = (1 \pm \epsilon)\lambda$ for all $i = 0, 1, \ldots, \lfloor \lg 1/p^* \rfloor$.

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \text{CONNECTIVITY}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

- *Analysis:*
  - Let λ_i be the size of min-cut of G_i
  - Karger's result implies $2^i \lambda_i = (1 \pm \epsilon)\lambda$ for all $i = 0, 1, \ldots, \lfloor \lg 1/p^* \rfloor$.
  - If $\lambda_i < k$, CONNECTIVITY_k algorithm guarantees $\lambda_i = \mu_i$.

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

- *Analysis:*
  - Let λ_i be the size of min-cut of G_i
  - Karger's result implies $2^i \lambda_i = (1 \pm \epsilon)\lambda$ for all $i = 0, 1, \ldots, \lfloor \lg 1/p^* \rfloor$.
  - If $\lambda_i < k$, $\textsc{connectivity}_k$ algorithm guarantees $\lambda_i = \mu_i$.
  - *Lemma:* $j \leq \lfloor \lg 1/p^* \rfloor$

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$
$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$
- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return
$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$
- *Analysis:*
  - Let λ_i be the size of min-cut of G_i
  - Karger's result implies $2^i \lambda_i = (1 \pm \epsilon)\lambda$ for all $i = 0, 1, \ldots, \lfloor \lg 1/p^* \rfloor$.
  - If $\lambda_i < k$, $\textsc{connectivity}_k$ algorithm guarantees $\lambda_i = \mu_i$.
  - *Lemma:* $j \leq \lfloor \lg 1/p^* \rfloor$
- Total space is $O(k \cdot n \cdot \text{polylog } n) = O(\epsilon^{-2} \cdot n \cdot \text{polylog } n)$.

Min-Cut Algorithm

- Let h_i be a hash function such that for each $e \in [n] \times [n]$

$$\mathbb{P}[h_i(e) = 1] = 1/2^i$$

- Let $G_i = (V, E_i)$ where $E_i = \{e \in E : h_i(e) = 1\}$
- Let $H_i = \textsc{connectivity}_k(G_i)$ where $k := 24\epsilon^{-2} \log n$
- Post-Processing: Let μ_i be min-cut size of H_i. Return

$$2^j \cdot \mu_j \quad \text{where } j = \min\{i : \mu_i < k\}$$

- *Analysis:*
  - Let λ_i be the size of min-cut of G_i
  - Karger's result implies $2^i \lambda_i = (1 \pm \epsilon)\lambda$ for all $i = 0, 1, \ldots, \lfloor \lg 1/p^* \rfloor$.
  - If $\lambda_i < k$, $\textsc{connectivity}_k$ algorithm guarantees $\lambda_i = \mu_i$.
  - *Lemma:* $j \leq \lfloor \lg 1/p^* \rfloor$
- Total space is $O(k \cdot n \cdot \text{polylog}\, n) = O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$.
- Can extend these ideas to get $(1 + \epsilon)$-sparsification of a dynamic graph in a single pass and $O(\epsilon^{-2} \cdot n \cdot \text{polylog}\, n)$ space.

Proof of Lemma

- Consider $i = \lfloor \lg 1/p^* \rfloor$ and so sampling probability for G_i is

$$2^{-i} < 2p^* = 12\lambda^{-1}\epsilon^{-2}\log n$$

Proof of Lemma

- Consider $i = \lfloor \lg 1/p^* \rfloor$ and so sampling probability for G_i is

$$2^{-i} < 2p^* = 12\lambda^{-1}\epsilon^{-2}\log n$$

- Consider a cut in G of size λ. Expected number of edges across same cut is G_i is at most

$$2p^* \cdot \lambda = 12\epsilon^{-2}\log n$$

and is $< \frac{24 \log n}{\epsilon^2} = k$ with high probability. Hence, $\lambda_i < k$.