CS 671: Graph Streaming Algorithms and Lower Bounds **Rutgers: Fall 2020**

Lecture 5

October 6, 2020

Instructor: Sepehr Assadi *Scribes: Milind Prabhu, Nirmit Joshi*

In this lecture, we will primarily focus on the following paper:

- "Kook Jin Ahn, Sudipto Guha, Andrew McGregor, Graph sketches: sparsification, spanners, and subgraphs. PODS 2012",

with additional background from:

- "Kook Jin Ahn, Sudipto Guha, Andrew McGregor, Analyzing Graph Structure via Linear Measurements. SODA 2012".

1 Dynamic Graph Streams

So far in the course, we have focused on *insertion-only* streams, i.e., when the edges of the graph are fixed a-priori and are being inserted one by one. However, many massive graphs of interest, say, social networks, are inherently dynamic; the links are formed and deleted over time. We will consider a variant of the graph streaming model for handling these scenarios, namely, the *dynamic graph streaming* model.

Let $G_0 = (V, E_0)$ be an empty graph on the vertex set $V := \{1, 2, \cdots, n\}$. A *dynamic stream* σ is a sequence of tuples $\langle\sigma_1, \sigma_2, \cdots, \sigma_t\rangle$ where each tuple $\sigma_i = (u_i, v_i, \Delta_i) \in [n] \times [n] \times \{-1, 1\}$ represents an update to the underlying graph G_i. The i-th update to G_i adds the edge (u_i, v_i) if $\Delta_i = 1$ and deletes (u_i, v_i) from the graph if $\Delta_i = -1$ to obtain the graph G_{i+1}. We assume that the stream deletes an edge only if it exists in the graph, i.e, there will not be a "negative" edge in the graph at any point. Let $G = (V, E)$ be the graph obtained after all the updates are performed. In the dynamic streaming model, we are interested in algorithms that make one or a few passes over σ (in the same order) and output solutions to problems on G. We will be sticking to the case where G is a simple undirected graph, although we can study more general (directed, multi) graphs in the same setting.

As before, the main objective in designing dynamic graph streaming algorithms is to minimize the space complexity and the number of passes required and then, to a lesser degree, the time complexity of the algorithm. Therefore, any problem can be solved using a single-pass over S and $O(n^2)$ space by just storing all the edges of G. Our main goal will be to see if we can answer any interesting questions about G using $o(n^2)$ space.

1.1 Finding an edge in the graph

Let us start by considering perhaps the simplest possible question here.

Problem 1. Design an algorithm that makes one pass over a dynamic stream σ of G and outputs any arbitrary edge from G.

We first begin by looking at a simpler version of Problem 1 where G is known to contain only one edge at the end of the stream. In this case, the following simple algorithm solves the problem: Consider any arbitrary mapping ϕ of unordered pairs of vertices to integers in $\left\{1, 2, \ldots, \binom{n}{2}\right\}$; maintain a counter c initialed at zero, and for any incoming tuple (u_i, v_i, Δ_i), let $c \leftarrow c + \Delta_i \cdot \phi(u_i, v_i)$. At the end, return $(u, v) = \phi^{-1}(c)$. It is straightforward to verify the correctness of the algorithm, and that it only requires $O(\log n)$ bits of space.

Can we extend this idea to the case when number of edges in G is arbitrary? The following simple lemma shows that this is effectively not possible without storing every edge of the graph.

Lemma 1. *Any deterministic algorithm that solves Problem 1, i.e., can find an edge in the graph (with n vertices) in the dynamic graph stream setting requires $\binom{n}{2}$ bits of space.*

Proof. Let G be any graph. Consider any deterministic algorithm that solves Problem 1 on G. Let $s(n)$ be the number of bits stored by the algorithm at the end of the stream. After the stream the algorithm outputs an edge e_1 in the graph. We now run the algorithm on the initial stream with an additional deletion of the edge e_1, namely, the graph $G - e_1$. The algorithm now outputs an edge e_2. We now run the algorithm on the initial stream with the deletions of e_1 and e_2, namely, the graph $G - e_1 - e_2$. We can repeat this process until we have deleted all the edges from the graph. This process also retrieves all the edges that were in G *using only* the $s(n)$ bits that were stored by the initial stream. Therefore the memory state at the end of the dynamic stream must be different for different input graphs. Since there are $2^{\binom{n}{2}}$ possible input graphs there must be at least $2^{\binom{n}{2}}$ different memory states. This implies that $s(n) \geq \binom{n}{2}$, proving the result. ☐

This lemma suggests that Problem 1 is surprisingly hard, but so far only for deterministic algorithms. The same sort of argument does not give us a lower bound for randomized algorithms. This is because the input to the randomized algorithm cannot be dependent on the random bits used by it – more formally, guarantee of a randomized algorithm only holds when we pick the random bits independent of the input; however, the above approach, once we delete the edge e_1 from the graph (which is a function of the random bits of the algorithm), there is no guarantee on the input graph $G - e_1$. This is not a coincidence: we will introduce a technique that will give us a randomized algorithm for Problem 1 which uses only $O(\text{poly}(\log n))$ space.

1.2 ℓ_0 Sampling

Consider the following problem.

Problem 2. Let $\boldsymbol{f} \in \mathbb{R}^n$ initially be $\mathbf{0}$. A sequence of updates are made to $\boldsymbol{f}$ and these updates appear in a stream $\sigma = \langle \sigma_1, \sigma_2, \cdots, \sigma_t \rangle$. Each $\sigma_j = (i, \Delta)$ corresponds to an update to $\boldsymbol{f}$ where f_i is updated by Δ, i.e, $f_i \leftarrow f_i + \Delta$. At the end of the stream we wish to sample an element f_j uniformly at random from the support of $\boldsymbol{f}$. We want to know both the index j and the value f_j of the sampled non-zero element.

It is easy to see that this problem is a generalization of Problem 1, over non-graph domains, and also when we require the algorithm to return a random element not an arbitrary one. The following theorem due to Jowhari, Salgam, and Tardos [7] gives an algorithm for this problem.

Theorem 2 ([7])**.** *There exists a linear sketch-based randomized algorithm that can perform ℓ_0 sampling on $\boldsymbol{f} \in \mathbb{N}^n$ with $\text{poly}(n)$-bounded entries using $O(\log^2(n) \log(1/\delta))$ space, and succeed with probability $\geq (1-\delta)$.*

There are a few things about this algorithm that we would like to emphasize.

- The algorithm is linear sketch-based which means that the only information it stores from the stream is $\mathcal{S} \cdot \boldsymbol{f}$ which is the matrix product of a $d \times n$ matrix $\mathcal{S}$ and $\boldsymbol{f}$. The matrix $\mathcal{S}$ is a "fat" matrix in that $d = O(\log n) \ll n$ and it is a function of the random bits supplied to the algorithm. The entries of $\mathcal{S}$ are computed whenever needed and $\mathcal{S}$ is not stored in the memory. Updates to $\boldsymbol{f}$ that arrive in the stream can directly be used to update $\mathcal{S}\boldsymbol{f}$. Specifically, if the update $\sigma_j = (i, \Delta)$ arrives in the stream, the algorithm updates $\mathcal{S}\boldsymbol{f} \leftarrow \mathcal{S}\boldsymbol{f} + \Delta\mathcal{S}^{(i)}$ where $\mathcal{S}^{(i)}$ is the i-th column of $\mathcal{S}$. To summarize, the ℓ_0 sampler only stores a few random linear projections of $\boldsymbol{f}$ that are easily updatable in the stream.
- An advantage of the algorithm being linear sketch-based is that ℓ_0 samplers $\mathcal{S}\boldsymbol{f_1}, \mathcal{S}\boldsymbol{f_2}$ of vectors $\boldsymbol{f_1}, \boldsymbol{f_2} \in \mathbb{R}^n$ can be used to find an ℓ_0 sampler of $\boldsymbol{f_1} + \boldsymbol{f_2}$ because $\mathcal{S}(\boldsymbol{f_1} + \boldsymbol{f_2}) = \mathcal{S}\boldsymbol{f_1} + \mathcal{S}\boldsymbol{f_2}$. We will see that this property can be exploited to a great extent in the sections to follow.

Note: From this point on, we will use $\mathcal{S}\boldsymbol{f}$ to denote the ℓ_0 sampler of $\boldsymbol{f}$ as described in Theorem 2.

It is easy to see that an ℓ_0 sampler can be used to find an edge in the graph G obtained at the end of a dynamic stream. Assign indices from 1 to $\binom{n}{2}$ to all possible edges in the graph and define $\boldsymbol{e} \in \{0,1\}^{\binom{n}{2}}$ as follows:

$$e_i = \begin{cases} 1 & \text{if the } i\text{-th edge is in } G \\ 0 & \text{otherwise} \end{cases}$$

The graph stream can be seen as a sequence of updates to $\boldsymbol{e}$. At the end of the stream the support of $\boldsymbol{e}$ will correspond to the edges that are present in the graph. An edge in the graph can be found by storing an ℓ_0 sampler $\mathcal{S}\boldsymbol{e}$. This requires at most $O(\log^2(n)\log(1/\delta))$ space and fails with probability at most δ. In the subsequent sections we will see clever applications of ℓ_0-samplers to dynamic graph streaming problems.

2 The Spanning Forest Problem

We now use ℓ_0-samplers to design a dynamic streaming algorithm for one of the most basic problems that we studied in insertion-only streams: finding a spanning forest of a given graph $G = (V, E)$.

2.1 Finding an edge crossing a cut

The first and highly important step here is to solve the following problem.

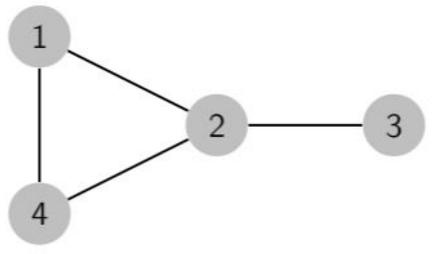
Problem 3. Find an algorithm that makes a single pass over a dynamic graph stream of graph G and finds an edge crossing a cut $(C, V \backslash C)$ in G. The cut $(C, V \backslash C)$ is revealed only at the end of the stream. The algorithm may only store $O(\text{poly}(\log n))$ bits of information per vertex.

To solve this problem, we first define a matrix representation of the graph.

Definition 3. Let $G = (V, E)$ be a simple undirected graph and A_G be an $n \times \binom{n}{2}$ matrix that is defined for this graph. Each column of A_G shall be indexed using a pair of numbers (j, k) where $j, k \in [n]$ and $j < k$. The (j, k)-th column of A_G corresponds to the edge (j, k) and the i-th row of A_G corresponds to vertex i in G. We will denote the i-th row of A_G by $\boldsymbol{a_i}$. The entries of the matrix are defined as follows:

$$a_{i,(j,k)} = \begin{cases} 1 & \text{if } i = j \text{ and } (j,k) \in E \\ -1 & \text{if } i = k \text{ and } (j,k) \in E \\ 0 & \text{if } (j,k) \notin E \end{cases}$$

See Figure 1 for an example.



$$A_G = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} (1,2) & (1,3) & (1,4) & (2,3) & (2,4) & (3,4) \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 \end{bmatrix} \end{array}$$

Figure 1: An illustration of the matrix representation of the graph.

The definition of A_G is motivated by the following lemma.

Lemma 4. *Let $(C, V \backslash C)$ be a cut in G and $\boldsymbol{x} = \sum_{i \in C} \boldsymbol{a_i}$. Then, $x_{(j,k)}$ is non-zero iff (j,k) is an edge in G crossing the cut.*

Proof. Consider some edge $e = (j,k)$ in G with $j < k$.

1. If j and k are both in C then $x_{(j,k)} = \sum_{i \in C} a_{i,(j,k)} = 1 + (-1) = 0$.
2. If j and k are both not in C then $x_{(j,k)}$ is 0 anyway.
3. If exactly one of j and k is in C and the other is in $V \backslash C$, i.e, the edge e crosses the cut, then $x_{(j,k)}$ is either 1 or -1.

We therefore conclude that $x_{(j,k)}$ is non-zero iff (j,k) crosses the cut. The lemma then follows. $\square$

Lemma 4 leads to the following algorithm to solve Problem 3.

Algorithm 1. Finding an edge crossing a cut

1. While processing the dynamic stream maintain n ℓ_0-samplers $\mathcal{S}\boldsymbol{a_1}, \mathcal{S}\boldsymbol{a_2}, \cdots, \mathcal{S}\boldsymbol{a_n}$, using the same sketching matrix $\mathcal{S}$ (with the same random bits across the vertices).
2. Let $(C, V \backslash C)$ be the input cut. Construct $\mathcal{S}\boldsymbol{x} = \sum_{i \in C} \mathcal{S}\boldsymbol{a_i}$ where $\boldsymbol{x} = \sum_{i \in C} \boldsymbol{a_i}$.
3. Find a non-zero element of $\boldsymbol{x}$ using the ℓ_0-sampler $\mathcal{S}\boldsymbol{x}$, and output the corresponding edge.

Algorithm 1 maintains an ℓ_0 sampler of size $O(\log^2 n)$ for each $\boldsymbol{a_i}$. The total space used by the algorithm is therefore $O(n \log^2 n)$ bits. The correctness follows from Lemma 4 and linear sketching property of ℓ_0-samplers. As such, Algorithm 1 gives us an algorithm for Problem 3. We now use this to solve the spanning forest problem.

Note: We will slightly abuse notation and refer to the ℓ_0 samplers $\mathcal{S}\boldsymbol{a_1}, \mathcal{S}\boldsymbol{a_2}, \cdots, \mathcal{S}\boldsymbol{a_n}$ together as $\mathcal{S}A_G$ in the future sections.

Note: We note that an important property of this algorithm is that the edge returned for the cut C is only a function of the sketches of the vertices in C. In particular, this means that if we have a collection of *vertex-disjoint* cuts $C_1, \ldots, C_k$, we can use the same algorithm to return one edge for every one of them in $O(n \log^2(n))$ bits.

2.2 Finding a spanning forest

Let us start by going over the classical Boruvka's algorithm [9] for finding spanning forest (we will then adapt this algorithm to dynamic streams).

Algorithm 2 (Boruvka's algorithm). A general algorithm to find a spanning forest

1. Initialize the set of supernodes to be the vertices of V.
2. In every *round*, while there exists an inter-supernode edge:
   (a) Find one edge incident on *every* supernode (and store the edges).
   (b) Contract connected supernodes into a single supernnode.
3. Output a spanning forest of stored edges.

We will now show that Algorithm 2 terminates in $O(\log n)$ rounds in a valid spanning forest of G.

Claim 5. *At any point of time, vertices in a supernode belong to the same connected component of G.*

Proof. We prove this by induction on the round number. At the beginning of the algorithm each supernode only contains a single vertex in the graph and the lemma holds. Let us say that the lemma holds after i rounds. During the $(i+1)$-th round consider a set of connected supernodes that are obtained after the addition of inter-supernode edges. It is easy to see that any two vertices belonging to these supernodes must be connected by a path. At the end of the round these supernodes are collapsed into a single supernode which must, by the previous observation, contain vertices from a single connected component. $\square$

The above claim implies that the vertices of any connected component $C \subseteq V$ of G at the end of any round i of Algorithm 2 are divided between a set of supernodes. Let $c_i(C)$ denote the number of supernodes corresponding to a connected component C at the beginning of round i.

Claim 6. *For any round i, If $c_i(C) > 1$ then $c_{i+1}(C) \leq \frac{1}{2} \cdot c_i(C)$.*

Proof. Let us look at how the number of supernodes corresponding to C changes after each round. During the i-th round, one edge incident on each supernode of C is found. Therefore, at the end of round i, each such supernode gets contracted to at least one other supernode. The number of new super nodes thus is getting halved, proving the claim. $\square$

The above claim then ensures that number of supernodes corresponding to a connected component gets halved in each round and is thus reduced to one after $O(\log n)$ rounds as desired.

Dynamic streaming implementation. We now adapt Algorithm 2 to find a spanning forest in dynamic streams. We will use $O(\log n)$ sketches $\mathcal{S}_i A_G$ that can be used to find an edge crossing a cut. We use $\mathcal{S}_1 A_G$ in the first round to find edges incident on each of the vertices. Next, we use $\mathcal{S}_2 A_G$ to find edges incident on supernodes in the second round. Note that $S_1 A_G$ cannot be used again in round 2 because that would mean that we have used S_1 in the first round to determine its own input in the second round. This is not allowed since the input of the randomized algorithm must be independent of its randomness. Therefore, we will need to use different sketches in different rounds.

Algorithm 3. Finding a Spanning Forest in Dynamic Streams

```
1. Sketch $A_G$ using matrices $\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_t$ where $t = O(\log n)$ as in Algorithm 1.
2. Initialize the set of supernodes $\hat{V}$ to be $V$.
3. For $r = 1$ to $t$ rounds:
    (a) Using $\mathcal{S}_r(A_G)$ try to find an edge incident on every supernode. If an edge is found, store it in
        the memory otherwise continue.
    (b) Collapse connected supernodes.
4. Output a maximal acyclic graph using the edges stored in memory.
```

Algorithm 3 stores $O(\log n)$ sketching matrices each of which has size $O(n \log^2 n)$ bits. Therefore storing all the sketching matrices requires $O(n \log^3 n)$ space. The algorithm also needs to store at most $O(n)$ edges at the end of each round. The number of edges stored across the $O(\log n)$ rounds is $O(n \log n)$ and the space required is $O(n \log^2 n)$ bits. So overall, the algorithm can find a spanning forest using only $O(n \log^3 n)$ space.

3 The k-Edge-Connectivity Problem

A simple undirected graph is said to be k-edge-connected if deleting fewer than k edges does not disconnect the graph. We show an extension of the previous algorithm to the edge-connectivity problem. First, we need the following graph theoretical result.

Proposition 7. *Consider a graph $G = (V, E)$ and integer $k \geq 1$. For $i \in [k]$, let F_i be spanning forest of $G \backslash (F_1 \cup F_2 \cup \cdots \cup F_{i-1})$. Then, $G = (V, E)$ is k-edge-connected iff $H = F_1 \cup F_2 \cup \cdots \cup F_k$ is k-edge-connected.*

Proof. If H is k-edge-connected this implies that G is k-edge-connected as H is a subgraph of G. It is enough to show that the k-edge-connectivity of G implies that k-edge-connectivity of H.

Assume for the sake of contradiction that G is k-edge-connected but H is not. This implies the existence of a cut $(C, V \backslash C)$ with at least k edges in G but less than k edges in H. Therefore, at least one of the k graphs F_i does not have an edge crossing the cut $(C, V \backslash C)$. Let F_j be the graph with the smallest index j that does not have an edge crossing the cut. Also let e be an edge crossing the cut that is in G but not in H. The edge $e \in G \backslash (F_1 \cup F_2 \cup \cdots \cup F_{j-1})$ but $e \notin F_j$. Since there is no edge crossing the cut $(C, V \backslash C)$ in F_j, e can be added to F_j without creating a cycle. Therefore, F_j is not a spanning forest of $(V, E \backslash F_1 \cup F_2 \cup \cdots \cup F_{j-1})$ which is a contradiction. □

The algorithm that we now describe creates spanning forests as described in Proposition 7 using Algorithm 3. It then checks if the union of these forests is k-edge-connected to examine the k-edge-connectivity of G.

Algorithm 4. k-EdgeConnect

1. Construct the sketches for k independent instatiations $\mathcal{I}_1, \mathcal{I}_2 \cdots, \mathcal{I}_k$ of the spanning forest algorithm (Algorithm 3).
2. For $i \in [k]$:
   (a) Use the sketches for $\mathcal{I}_i$ to find a spanning forest F_i of $G \backslash (F_1 \cup F_2 \cup \cdots \cup F_{i-1})$.
   (b) Update the sketches $\mathcal{I}_{i+1}, \mathcal{I}_{i+2}, \cdots, \mathcal{I}_k$ by deleting all edges of F_i (this can be done using linearity of sketches).
3. Test whether $H = F_1 \cup F_2 \cup \cdots \cup F_k$ is k-connected and return H.

The correctness of this algorithm follows immediately from Proposition 7 and correctness of Algorithm 3. The space complexity is also k times that of Algorithm 3, and thus is $O(kn \cdot \log^3(n))$.

4 The (Approximate) Minimum Cut Problem

Finally, we give a dynamic streaming algorithm for finding a $(1 + \varepsilon)$-approximation to the minimum cut problem. Recall that in the minimum cut problem, the goal is to find a cut with a minimum number of edges, or alternatively, the minimum number of edges whose removal would disconnect the graph. To continue, we first need some notation and preliminaries.

4.1 Graph Definitions and Preliminaries

Given an undirected unweighted graph $G(V, E)$, we define:

- $\lambda(G)$: the size of a minimum cut of G, i.e.,
- $\lambda_{u,v}(G)$: the size of a minimum u-v cut in G for any $u, v \in V$, and,

- $\lambda_A(G)$: the size of the cut $(A, V \setminus A)$ for any non-empty $A \subset V$.

A key definition for us is that of *sparsifier*. Roughly speaking, rather than just determining whether a graph is connected, these sparsifiers allow us to estimate a richer set of connectivity properties such as the size of *all* cuts in the graph. In general, there are different types sparsifiers. But we will stick to the following definition of *cut sparsifier* introduced by Benczúr and Karger [4].

Definition 8 (Cut Sparsifier). Given a graph $G(V, E)$, we say that a weighted subgraph $H = (V, H, w)$ is an ε-sparsifier for G if for all non-empty $A \subset V$, the following holds:

$$(1-\varepsilon)\cdot\lambda_A(G) \leq \lambda_A(H) \leq (1+\varepsilon)\cdot\lambda_A(G).$$

Cut Sparsifiers via Karger's Uniform Sampling. Perhaps the simplest way to achieve a cut sparsifier is via a uniform sampling approach due to Karger [8]. Although by now there are much more efficient methods and better sparsifiers (in terms of number of edges in the sparsifier) are known, for our purpose in this lecture, this simple approach suffices.

Let H be a weighted subgraph of G obtained by sampling each edge of G independently with probability p and setting the weight of each sample edge to be $1/p$. We prove that such a graph provides an ε-sparsifier of G for a sufficiently large value of p (as a function of ε). See Figure 2 for an illustration.

Lemma 9. *In an undirected unweighted graph G, if we sample each edge with probability $p \geq \min\{\frac{6\log n}{\lambda\varepsilon^2}, 1\}$ and assign the weight $1/p$ to sampled edges, then the resulting graph H is an ε-sparsifier of G with high probability.*

Proof. Each edge is sampled *independently* with probability p. If the edge survives then it contributes $1/p$ to the size of any cut it crosses, otherwise it just contributes 0. Thus, in expectation, each edge still contributes 1 to the size of any cut it crosses. Therefore, for any cut $(A, V \setminus A)$, $\mathbb{E}[\lambda_A(H)] = \lambda_A(G)$.

Fix any cut A in G and let $\lambda_A(G) = \alpha\lambda$ for some $\alpha \geq 1$.

$$\begin{aligned}
\Pr\left(\lambda_A(H) \notin (1\pm\varepsilon)\cdot\lambda_A(G)\right) &= \Pr\left(\#\text{ of sampled edges crossing the cut} \notin (1\pm\varepsilon)\cdot p\cdot\lambda_A(G)\right) \\
&\leq \exp\left(\frac{-\varepsilon^2\cdot p\cdot\lambda_A}{2}\right) = \exp\left(\frac{-\varepsilon^2\cdot p\cdot\alpha\cdot\lambda}{2}\right) \qquad \text{(Using Chernoff Bound)} \\
&\leq \frac{1}{n^{3\alpha}}.
\end{aligned}$$

We can now use the following well-known graph theory result whose proof we omit.

Proposition 10. *There are at most $n^{2\alpha}$ cuts with size at most $\alpha\lambda$ in any graph with min cut value λ.*

Using Proposition 10 and union bound, we upper bound the probability of the event that some cut is not preserved in the constructed graph.

$$\Pr\left(\text{There exists } (A, V\setminus A) \text{ s.t. } \lambda_A(H) \notin (1\pm\varepsilon)\cdot\lambda_A(G)\right) \leq \sum_{\alpha=1}^{n^2} n^{-3\alpha}\cdot n^{2\alpha} \leq \sum_{\alpha=1}^{\infty}\frac{1}{n^{\alpha}} = O(1/n).$$

Thus, we can conclude that all the cuts are preserved and the resulting graph after doing Karger's construction is an ε-sparsifier of the original graph with high probability. □

Now recall that our main purpose behind sampling edges in a graph was that it may sparsify the graph. However, the sampling scheme suggested by Karger does not always find a "small" sparsifier, say, with only $\tilde{O}(n)$ edges. In fact, it is very easy to come up with an example in which Kerger's sampling finds a dense sparsifier for us, which has $\Omega(n^2)$ edges. Consider a clique of size $n-1$ connected to one other vertex by a bridge edge. In this case, the Karger's sampling simply samples every edge in the graph with probability 1.

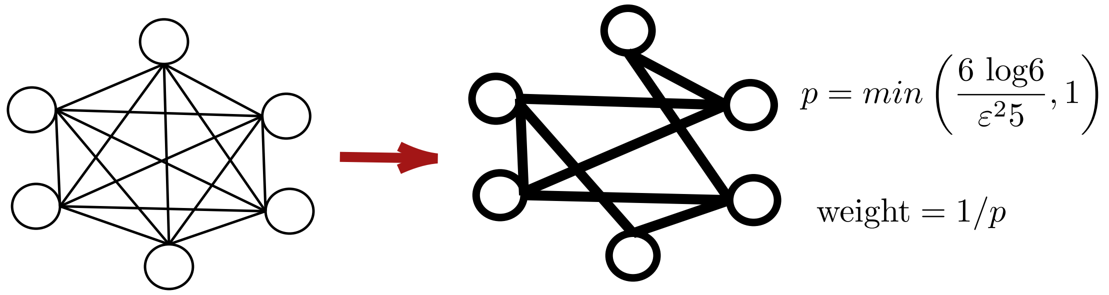


Figure 2: The transformation of a complete graph on 6 vertices after Karger's uniform sampling. The darker edges represent the increased edge weights from 1 to $1/p$.

Also, not knowing λ upfront hinders the possibility of adapting this algorithm in the streaming setting since p is unknown. As stated earlier, there known sparsifiers with $\tilde{O}(n/\varepsilon^2)$ edges [4] and in fact even $O(n/\varepsilon^2)$ edges [3] (which is also known to be optimal [2, 5]) – however, for our purpose in this lecture, we can stick with this simplest version and show how to address all these challenges.

5 A Dynamic Streaming Algorithm for Approximate Min Cut

Let us first formally define the problem in the dynamic streaming setting.

Problem 4. Given an approximation parameter $\varepsilon \in (0, 1)$ and a graph $G(V, E)$, where the edges in E come in a dynamic stream, find approximated value of the min-cut $\tilde{\lambda}$ such that the following holds:

$$(1 - \varepsilon) \cdot \lambda(G) \le \tilde{\lambda} \le (1 + \varepsilon) \cdot \lambda(G).$$

We study an algorithm due to Ahn, Guha and McGregor [1] that solves Problem 4. Roughly speaking, the algorithm first computes a sequence of graphs $G = G_0 \supseteq G_1 \supseteq G_2 \supseteq \dots$, where each G_i is formed *on the fly* by *independently* removing each edge in G_{i-1} with probability 1/2. See Figure 3. Thus, any edge survives in G_i with probability $1/2^i$. The algorithm uses uniform hash functions to do this. The algorithm then runs k-EdgeConnect (Algorithm 4) simultaneously on each subgraph G_i, to construct a sequence of witness graphs $H_0, H_1, H_2, \dots$. Formally,

Algorithm 5. A semi-streaming algorithm to approximate the min-cut size in dynamic streams.

1. For $i \in \{1, \dots, 2\log n\}$, let $h_i : E \to \{0, 1\}$ be a uniform hash function.
2. For $i \in \{0, 1, \dots, 2\log n\}$:
   - Let G_i be the subgraph of G containing edges e s.t. $\prod_{j \le i} h_j(e) = 1$.
   - Let $H_i \leftarrow k$-EdgeConnect(G_i) for $k = 24 \cdot \varepsilon^{-2} \cdot \log n$.
3. Return $\tilde{\lambda} := 2^j \cdot \lambda(H_j)$, where $j = \min\{i : \lambda(H_i) < k\}$.

As it can be seen in Algorithm 5, we just scale the min-cut size of the first subgraph in the sequence with size less than k and return it. The main intuition behind this step is that k-EdgeConnect algorithm preserves all the cuts of G_i having size less than k. And G_i preserves all the cuts of G on appropriate scaling, including the min-cut, if the sampling probability is high enough due to Lemma 9. The idea is that if i is not too large, $\lambda(G)$ can be approximated via $\lambda(G_i)$ and if $\lambda(G_i) < k$ then $\lambda(G_i)$ can be calculated from H_i. Now, we formalize this in the following and show the correctness.

Let λ denote the min-cut size in G and $p := \min\{\frac{6\log n}{\lambda\varepsilon^2}, 1\}$. Moreover, we let i^* be the largest index where

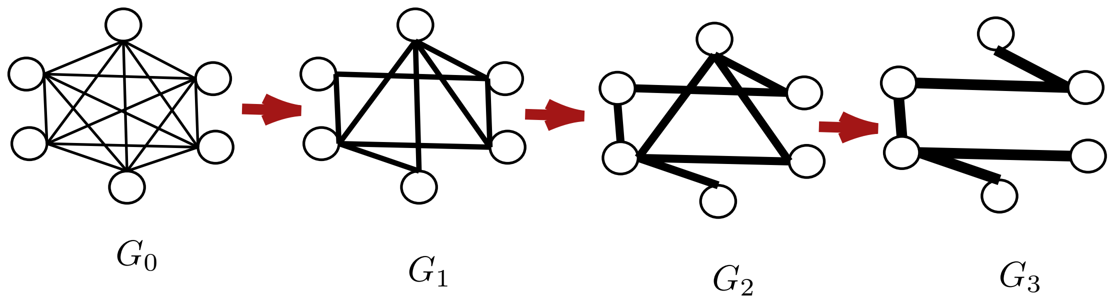


Figure 3: A sequence of subgraphs created by Algorithm 5.

each edge is sampled with probability at least p. Therefore,

$$\frac{1}{2^{i^*}} \geq p > \frac{1}{2^{i^*+1}}, \text{ which implies, } i^* = \lfloor \log_2(1/p) \rfloor.$$

Now, by Lemma 9, for $i \leq i^*$, if we set all the edge weights in G_i to be 2^i, then G_i approximates all the cut values in G w.h.p.

Lemma 11. *In G_{i^*}, the min-cut size $\lambda(G_{i^*})$ is less than k with high probability.*

Proof. We know that each edge of G remains in G_{i^*} with probability $1/2^{i^*}$ independently. By linearity of expectation,

$$\mathbb{E}\left[\lambda(G_{i^*})\right] = \lambda/2^{i^*} \leq 2 \cdot \lambda \cdot p \leq 12 \log n \cdot \varepsilon^{-2}.$$

Now, we upper bound the probability of the event: $\lambda(G_{i^*}) \geq k$.

$$\begin{aligned}
\Pr\left(\, \lambda(G_{i^*}) \geq k \,\right) &= \Pr\left(\, \lambda(G_{i^*}) \geq \frac{24 \log n}{\varepsilon^2} \,\right) \leq \Pr\left(\, \lambda(G_{i^*}) \geq 2\,\mathbb{E}\left[\lambda(G_{i^*})\right] \,\right) \\
&\leq \Pr\left(\, |\lambda(G_{i^*}) - \mathbb{E}\left[\lambda(G_{i^*})\right]| \geq \mathbb{E}\left[\lambda(G_{i^*})\right] \,\right) \\
&\leq 2 \cdot \exp\Big(-\frac{\mathbb{E}\left[\lambda(G_{i^*})\right]}{3}\Big) && \text{(By Chernoff Bound)} \\
&\leq 2 \cdot \exp\Big(-\frac{\lambda \cdot p}{3}\Big) \leq 2 \cdot \exp\big(-2 \log n\big) \\
&= O(1/n^2),
\end{aligned}$$

concluding the proof. □

The above lemma allows us to show the correctness of Algorithm 5. In particular, we prove the following theorem:

Theorem 12. *Assuming access to fully independent random hash functions, there exists a single-pass, $O(\varepsilon^{-2} \cdot n \cdot \log^5 n)$-space algorithm that $(1 \pm \varepsilon)$-approximates the minimum cut with high probability in the dynamic graph stream model.*

Proof. We run k-EdgeConnect algorithm in parallel $O(\log n)$ times, once on each subgraph G_i. Each execution takes $O(k \cdot n \cdot \log^3 n)$ space. Hence, the total space used by Algorithm 5 is $O(k \cdot n \cdot \log^4 n) = O(\varepsilon^{-2} \cdot n \cdot \log^5 n)$ since $k = O(\varepsilon^{-2} \log n)$. It thus only remains to show that the algorithm approximates the min-cut, which we do in the following.

If a cut in G_i has less than k edges that cross the cut, the witness contains all such edges. On the other hand, if a cut value is larger than k, the witness contains at least k edges that cross the cut. Therefore, if G_i is not k-edge-connected, we can correctly find a minimum cut in G_i using the corresponding witness.

And by Lemma 11, we know that the number of edges in G_{i^*} that crosses the minimum cut of G is less than k with high probability. Moreover, for all $i \leq i^*$, on setting all the edge weights in G_i to 2^i, G_i approximates all the cut values in G w.h.p. by Lemma 9. Therefore, Algorithm 5 returns the min-cut size to a $(1 \pm \varepsilon)$-approximation. □

5.1 Pseudo Random Generators (PRGs)

Throughout the discussion, including Theorem 12, we assumed the access to *fully independent* uniform hash functions. But implementing these hash functions takes up to $O(m \log n)$ space. This indeed is a lot of space! We conclude this section by pointing out a standard method for fixing this issue using *pseudo random-number generators* (PRGs). This argument is originally due to Indyk [6] and was adapted by Ahn, Guha, and McGregor [1] for the purpose of this specific algorithm.

The key idea is to use Nisan's PRG [10] in the following result.

Proposition 13 (Nisan's PRG [10])**.** *Any randomized algorithm running in space S and using R random bits may be converted to one that uses only $O(S \log R)$ random bits and $O(S \log R)$ space, while increasing the failure probability by at most $2^{-\Theta(S)}$.*

The proof of this result is well beyond the scope of this course. We now show how to use this result for our purpose in dynamic streaming algorithm.

As described, the space of our algorithm is $O(n \cdot \text{poly}\log(n))$ assuming access to fully independent random hash functions; however, we also need $O(n^2 \cdot \text{poly}\log(n))$ space to store these hash functions and thus the overall space of the algorithm is actually $S = O(n^2 \cdot \text{poly}\log(n))$. As such, it is not immediately clear how we can apply Proposition 13. However, consider the following: suppose our algorithm is used on a *sorted edge stream* where all insertions and deletions for a single edge come in consecutively. In this case, at any given time, we only need to store one random bit for each hash function, which requires just $O(\log n)$ space. The random bits can be discarded after moving on to the next edge. Thus, the entire algorithm can run in $O(n \cdot \text{poly}\log(n))$ space over these types of streams. Then, we can apply Proposition 13, using the PRG to get all of our required random bits by spending $O(S \log R) = O(n \cdot \text{poly}\log(n))$ truly random bits. This only increases the error probability by $\ll 2^{-n}$ which is negligible.

Now notice that, since our algorithm is sketch based, edge updates simply require an addition to or subtraction from a sketch matrix. These operations commute, so our output will not differ if we reorder the insertions and deletions in the stream. Thus, we can run our algorithm on any general edge stream, use PRG to generate any of the required $O(n^2 \cdot \text{poly}\log(n))$ bits and operate in only $O(n \cdot \text{poly}\log(n))$ space. Each time an edge is streamed in, we need to generate $O(\log n)$ random bits from the PRG, which can be done by going over the randomness of input to PRG from scratch and recomputing the value of desired hash functions based on the PRG.

As a result, we can conclude the following theorem of Ahn, Guha, and McGregor [1].

Theorem 14 ([1])**.** *There exists a single-pass, $O(\varepsilon^{-2} \cdot n \cdot \text{poly}\log(n))$-space algorithm that $(1 \pm \varepsilon)$-approximates the minimum cut with high probability in the dynamic graph stream model.*

References

[1] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 5–14. ACM, 2012. 8, 10

[2] A. Andoni, J. Chen, R. Krauthgamer, B. Qin, D. P. Woodruff, and Q. Zhang. On sketching quadratic forms. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 311–319, 2016. 8

[3] J. D. Batson, D. A. Spielman, and N. Srivastava. Twice-ramanujan sparsifiers. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 255–262, 2009. 8

[4] A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In G. L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 47–55. ACM, 1996. 7, 8

[5] C. Carlson, A. Kolla, N. Srivastava, and L. Trevisan. Optimal lower bounds for sketching graph cuts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2565–2569, 2019. 8

[6] P. Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006. 10

[7] H. Jowhari, M. Sağlam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 49–58, 2011. 2

[8] D. R. Karger. Random sampling in cut, flow, and network design problems. In F. T. Leighton and M. T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 648–657. ACM, 1994. 7

[9] J. Nesetril, E. Milková, and H. Nesetrilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discret. Math.*, 233(1-3):3–36, 2001. 4

[10] N. Nisan. Pseudorandom generators for space-bounded computation. *Comb.*, 12(4):449–461, 1992. 10