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

Lecture 11

November 17, 2020

Instructor: Sepehr Assadi *Scribe: Shreya Mogulothu and Vihan Shah*

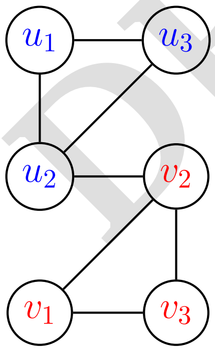
1 Minimum cut in Streaming

1.1 Introduction

In a graph $G = (V, E)$, a cut $C \subset V$ is a set of vertices of G. The edges of a cut $E(C)$ is a set containing the edges going out of C i.e. the edges that have one end point in C and another in $V \setminus C$. The size of the cut $|E(C)|$ is the number of edges going out of C and the notation for it is $\lambda_C(G)$ or just λ_C if it is clear what graph we are talking about. The minimum cut problem asks to find a cut $C \subset V$ such the the size of the cut is minimized over all possible cuts. Observe that the minimum degree d of the graph is an upper bound on the min cut size since a single vertex also forms a cut called a singleton cut or a trivial cut. The minimum cut problem is of great interest and has numerous applications.

Definition 1. The minimum cut of a graph $G = (V, E)$ is the minimum size of a cut, over all cuts of G i.e. $\min_{C \subset V} |E(C)|$.

This example shows that the min degree is not always equal to the min cut value. Here the min degree $d = 2$ but the min cut $c = 1$. The min cut $C = \{u_1, u_2, u_3\}$. In general the difference between the min degree and the min cut can be made very large by connecting two cliques with an edge.



Another interesting graph problem is the edge-connectivity problem. A connected graph is k-edge connected if it remains connected whenever fewer than k edges are removed. The edge-connectivity of a graph is the largest k for which the graph is k-edge connected.

Claim 2. *For any graph $G = (V, E)$ size of the min cut c is equal to the edge-connectivity k.*

Proof. We first show that $k \leq c$. Let C be the cut that has size c. Thus there are c edges that have one end point in C and another in $V \setminus C$. If we delete these c edges then there are no edges between C and $V \setminus C$ thus the graph is disconnected implying $k \leq c$.

Now we show $c \leq k$. Let $e_1, e_2, \dots e_k$ be the k edges that when deleted, disconnect G. After deleting these edges we should have two connected components S and $V \setminus S$. If there are more than two connected components then adding one edge back keeps G disconnected implying edge connectivity $< k$. These two components S and $V \setminus S$ correspond to a cut in G thus $c \leq k$. $\square$

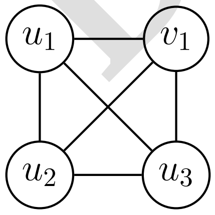
Edge connectivity is a fundamental measure for robustness of graphs. Many techniques have been used to find the edge connectivity including max flow computation, maximum adjacency ordering, random contraction etc. The fastest known deterministic algorithm is by Henzinger, Rao, and Wang [4] with a runtime of $O(m \log^2 n \log\log n)$ using a local flow technique. The fastest randomized algorithm is by Ghaffari, Nowicki, and Thorup [3] with a runtime of $\min\{O(m \log n), O(m + nlog^2 n)\}$ using a contraction technique. This algorithm achieves the given runtime due to a recent improvement by Gawrychowski, Mozes, and Weimann in finding the min cut in a weighted graph. A simple deterministic algorithm was given by Thatchaphol [6] with runtime $O(m^{1+o(1)})$ using expander decomposition.

In the streaming setting, Zelke showed a lower bound of $\Omega(n^2)$ for one pass streaming algorithms using the communication complexity of the index problem [7]. Ahn and Guha provided a one pass streaming algorithm for a $(1 \pm \varepsilon)$ sparsifier using $\tilde{O}(n/\varepsilon^2)$ space [1]. The problem of finding the exact min cut in multiple passes was open until Rubinstein, Schramm, and Weinberg gave an algorithm in cut-query model [5]. [2] observed that it became a two-pass semi-streaming immediately. [AD21] gave a simpler and more direct algorithm very recently.

1.2 High Level idea

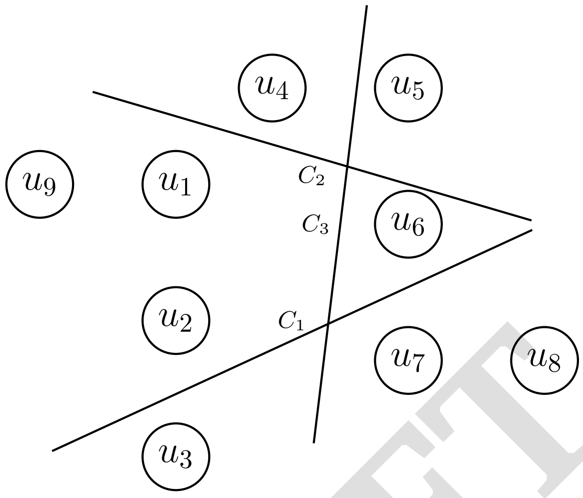
We would like to find the min cut of the graph. The problem could be made slightly easier if we only focus on non trivial cuts since it is easy to find the minimum trivial cut which is the minimum degree. In the first pass we compute an ε sparsifier of the graph. Also, in parallel, we compute the min degree of the graph. The sparsifier gives us a good idea of which cuts are big and which ones are small thus we can compress the big cuts of the graph. We can show that this compressed graph has only $O(n)$ edges thus we can store the whole compressed graph in another pass. Thus in the second pass we store the entire compressed graph. The graph was compressed in a way to store all small non trivial cuts including non trivial min cut. Thus we can finally find the min cut by taking the minimum of the min cut of the compressed graph and the min degree. Note that we can output the cut as well as the size of the cut by storing the vertex having minimum degree and the supernodes that form the minimum cut in the compressed graph.

Note that it is important to deal with the trivial cuts separately. If try to store the trivial cuts as well in the compressed graph, we might get too many edges in our compressed graph. If we consider a clique we see that there is no way we can compress it if we want to preserve the trivial min cuts. Lemma 5 tells us that trivial cuts behave very differently from non trivial cuts.



1.3 First pass

Let the graph in the stream be $G = (V, E)$ with min cut value c and min degree d. Assume that there exists a blackbox algorithm to get a $1 \pm \delta$ sparsifier of G. Run that algorithm in the first pass to get a $1 \pm \delta$ sparsifier H of G. Also, in parallel, compute the min degree of G. Now we use H to identify the ε approximate min cuts of G where $\varepsilon = \frac{(1+\delta)^2}{(1-\delta)^2} - 1$. We would like to compress the large cuts of G and ensure that only the small ones survive in the compressed graph. The guarantees we should have are that all non trivial min cuts survive the contraction and all cuts that survive the contraction are ε approximate min cuts. Let $\hat{c}$ be the value of the min cut of H (considering both trivial and non trivial cuts). Let $\hat{\mathcal{C}}$ be the set of all non trivial cuts with value less than or equal to $\frac{(1+\delta)}{(1-\delta)}\hat{c}$. We will show that $\hat{\mathcal{C}}$ is the set of cuts that satisfies both the properties we wanted. Therefore we can use $\hat{\mathcal{C}}$ create the compressed graph. All the nodes that lie on the same side of all cuts in $\hat{\mathcal{C}}$ belong to the same supernode in the compressed graph G'. An easy way to find these supernodes is the following. Start with a single partition V. Go over the cuts in $\hat{C}$ one by one and for each cut C divide all the current partitions into two based on which side of cut C they lie on. It could be possible that a partition is not divided into two but just stays as one partition. Thus in this procedure we start with V and then get two partitions and then more. Once all the cuts in $\hat{\mathcal{C}}$ have divided the partitions, we are left with l partitions which will be the super nodes of G' (all nodes in one partition will be part of one supernode).



Claim 3. *$\hat{\mathcal{C}}$ contains all non trivial min cuts with value c.*

Proof. Consider any non trivial min cut C in G. We know that $\lambda_C(H) \leq (1+\delta)c$ and also $(1-\delta)c \leq \hat{c}$. Thus,

$$\begin{aligned}\lambda_C(H) &\leq (1+\delta)c \\ &= \frac{(1+\delta)}{(1-\delta)}(1-\delta)c \\ &\leq \frac{(1+\delta)}{(1-\delta)}\hat{c}.\end{aligned}$$

Therefore C will be included in $\hat{\mathcal{C}}$. □

Claim 4. *All cuts in $\hat{\mathcal{C}}$ are ε approximate min cuts.*

Proof. In other words, any cut C which is an ε' approximate min cut will never be a part of $\hat{\mathcal{C}}$ for $\varepsilon' > \varepsilon$. We know that $(1-\delta)(1+\varepsilon')c \leq \lambda_C = (1+\varepsilon')c$ and $\hat{c} \leq (1+\delta)c$. Thus,

$$\begin{aligned}\lambda_C &= (1+\varepsilon')c \\ &\geq (1-\delta)(1+\varepsilon')c \\ &> (1-\delta)(1+\varepsilon)c \\ &= \frac{(1+\delta)}{(1-\delta)}(1+\delta)c && \text{(Substituting } 1+\varepsilon) \\ &\geq \frac{(1+\delta)}{(1-\delta)}\hat{c}.\end{aligned}$$

All the cuts in $\hat{\mathcal{C}}$ are ε approximate min cuts because larger cuts are not accepted in $\hat{C}$. □

We have shown that $\hat{\mathcal{C}}$ satisfies both the properties we wanted. Now consider the following lemma that we will prove later.

Lemma 5. *Let $G = (V, E)$ be an unweighted graph with minimum degree d and minimum cut value c. Let $\mathcal{C}$ be the set of all non-trivial cuts in the graph, with cut value at most $c + \varepsilon d$, for $\varepsilon < 1$. Then $|\cup_{C \in \mathcal{C}} E(C)|$ (the total number of edges that participate in cuts in C) is $O(n)$.*

The above lemma will help us bound the number of edges in $\hat{C}$.

Claim 6. *Number of edges in $\hat{\mathcal{C}}$ is $O(n)$ i.e. $|\cup_{C \in \hat{\mathcal{C}}} E(C)| = O(n)$.*

Proof. Lemma 5 states that $|\cup_{C \in \mathcal{C}} E(C)| = O(n)$. We will show $\hat{\mathcal{C}} \subseteq \mathcal{C}$ implying $|\cup_{C \in \hat{\mathcal{C}}} E(C)| = O(n)$. Any cut C in $\hat{\mathcal{C}}$ is an ε approximate min cut by Claim 4 thus it has at most $(1 + \epsilon)c \leq c + \varepsilon d$ edges (since min degree is an upper bound on c). Also, C is a non trivial cut by definition of $\hat{C}$. Thus $C \in \mathcal{C}$ because it has at most $c + \varepsilon d$ edges and it is a non trivial cut. Therefore $\hat{\mathcal{C}} \subseteq \mathcal{C}$. ☐

1.4 Second Pass

After the first pass, we store the super nodes of the compressed graph G'. Thus for each node in G we store which supernode it belongs to. In the second pass we store the graph G' exactly. For an edge (u, v) in the stream we check if u, v belong to the same supernode. If yes then we do not add the edge to G' and otherwise we do. After the second pass we have the graph G' and we compute its min cut C' with value c'. We compare this min cut value c' with the stored min degree d and output the minimum of the two. The minimum will be the min cut of G i.e. $\min(d, c') = c$. To output the cut vertices of the min cut we output the vertex with the min degree if d is smaller and we output the vertices in the supernodes of cut C' if c' is smaller.

Correctness:

If the min cut is trivial then the min degree has value c thus $min(c', d) = d = c$. If the min cut C is non trivial then we know that $C \in \hat{\mathcal{C}}$ thus all the edges of C are preserved in G'. Also, any node on one side of C can not be in the same supernode as a node on another side of C because they do not lie on the same side of every cut in $\hat{\mathcal{C}}$ $(C \in \hat{\mathcal{C}})$. Thus C is preserved in G' and $min(c', d) = c' = c$. Also, ever cut in G' corresponds to a cut in G so we can never get a cut value smaller than c. Thus we will always output c.

Space:

We set $\varepsilon = 0.08$ thus $\delta = 0.02$. G' only contains the edges that are in $\hat{\mathcal{C}}$. This is because for any edge (u, v) not in $\hat{\mathcal{C}}$, u and v are on the same side of all cuts in $\hat{\mathcal{C}}$ and thus belong to the same supernode. Thus storing G' only takes $O(n)$ space since ε is a constant. Storing the min degree in the first pass can be done in $\tilde{O}(n)$ space by storing the degree of each node and computing the minimum after the stream. Also, the sparsifier takes $\tilde{O}(n)$ space when δ is a constant. Thus the total space is $\tilde{O}(n)$.

2 Proof of Lemma 5

2.1 Setting up the Proof

For any $\varepsilon > 0$, let $0 < \alpha + \beta < \frac{1}{2}(1 - \varepsilon)$.

Notice that any subset of $\mathcal{C}$ induces a partition over V, where two vertices are in the same component if they are on the same side of every cut. Let $\mathcal{K}$ be a subset of $\mathcal{C}$ chosen as follows: starting from $\mathcal{K}$ empty, while there exists a cut $C \in \mathcal{C}$ such that adding C to $\mathcal{K}$ splits at least one component into two components each of size $\geq \beta d$, add C to $\mathcal{K}$. At termination, $\mathcal{K}$ contains k cuts $\mathcal{K} = C_1, \ldots, C_k$.

Bounding k

Claim 7. *The union of cuts in $\mathcal{K}$ contains at most $(c+\varepsilon d)k$ edges: $|\cup_{i\in[k]} E(C_i)| \le (c+\varepsilon d)k$.*

Proof. Each of the k cuts is an approximate min cut and contains at most $c+\varepsilon d$ edges. □

Claim 8. $k \le \frac{n}{\beta d} - 1$.

Proof. A component of the partition $\mathcal{K}$ is directly charged at time t if adding C_t splits component into two components of size $\ge \frac{n}{\beta d}$ and indirectly charged at time $t' > t$ if one of its descendent components is directly charged at time t'.

We prove by induction that any set of size x is charged at most $\frac{x}{\beta d}$ times. For the base case, we take $x = 2\beta d$. A component of this size can be charged at most once, because it can be charged directly if it is split into two sets of size exactly βd, but never charged again. Now, suppose that the claim holds for any $y < x$, and consider a set X of size x. The first time X is split, it becomes refined into two sets U, W of sizes u and w. The set X may be charged during this split. By induction, U may be charged at most $\frac{u}{\beta d} - 1$ times, and W may be charged at most $\frac{w}{\beta d} - 1$ times. So X can be charged at most $(\frac{u}{\beta d} - 1) + (\frac{w}{\beta d} - 1) + 1 = \frac{x}{\beta d} - 1$ times, as desired.

To finish the proof, we note that the set of all nodes is charged at most $\frac{n}{\beta d} - 1$ times, and every cut C_i causes the set of all nodes to be charged. □

Misrepresented Edges

Now, define S to be the set of vertices $v \in V$ with at least $\alpha \deg(v)$ edges in $\mathcal{K}$. We claim that S cannot be too large:

Claim 9. $\sum_{v\in S} \deg(v) \le \frac{4dk}{\alpha}$

Proof. There are at most $(c+\varepsilon d)k \le 2dk$ edges in $\mathcal{K}$. Therefore, there are at most $4dk$ edge, vertex pairs for which the edge is in $\mathcal{K}$ and is incident to some vertex. By definition, any vertex $v \in S$ participates in at least $\alpha \deg(v)$ such pairs, which implies that $\alpha \sum_{v\in S} deg(v) \le 4dk$. □

Define v is small for cut C if adding C to $\mathcal{K}$ causes v's component to shrink to size less than βd. Define v small for $\mathcal{K}$ if v is small for some cut $C \notin \mathcal{K}$.

Claim 10. *If v is small for $\mathcal{K}$ then $v \in S$.*

Proof. Assume for contradiction that $v \notin S$, and let C denote the cut for which v is small. Let B be the new connected component of v if we were to add C to K. Then v has fewer than βd neighbors in B, because v is small and therefore $|B| < \beta d$. Since $v \notin S$, v has at most $\alpha deg(v)$ edges incident in K. Therefore, v has at least $deg(v) - \alpha deg(v) - \beta d$ incident edges crossing the cut C.

Since C is not a singleton cut, we can move v to the other side of C, and decrease its size by

$$\begin{aligned}\deg(v) - \alpha\deg(v) - \beta d - (\alpha\deg(v) + \beta d) &= \deg(v) - 2\alpha\deg(v) - 2\beta d \\ &\le \deg(v)(1 - 2\alpha - 2\beta) \\ &\le \deg(v)\cdot\varepsilon \\ &\le \varepsilon d\end{aligned}$$

since $2(\alpha+\beta) < 1 - \varepsilon$. This is a contradiction, since we cannot have a cut of size $< c$. □

Claim 11. *If an edge not in $\mathcal{K}$ belongs to some cut C, then at least one of its vertices is small.*

Proof. Since (u, v) is not in $\mathcal{K}$, u and v are in the same component. Also, since we did not add C to $\mathcal{K}$, it must be the case that one of u or v is small for C and therefore small for $\mathcal{K}$ and therefore included in S. $\square$

We have that every edge in $|\cup_{C\in\mathcal{C}} C|$ is either in $\mathcal{K}$ or incident to a vertex in S. We have that

$$\begin{aligned} |\cup_{C\in\mathcal{C}}| &\leq |\cup_{C\in\mathcal{K}} C| + \sum_{v\in S} deg(v) \\ &\leq 2dk + \frac{4dk}{\alpha} \\ &\leq 2d \cdot \frac{n}{\beta d} + \frac{4d}{\alpha} \cdot \frac{n}{\beta d} \\ &= O(n) \end{aligned}$$

where we have first applied our bounds on the volume of S and the number of edges in $\mathcal{K}$, then applied our bound on k. The conclusion follows.

3 Sparsifier Construction

Definition 12. An ε cut sparsifier of a graph G is a graph H such that for every cut C in G we have

$$(1-\varepsilon)\lambda_C(G) \leq \lambda_C(H) \leq (1+\varepsilon)\lambda_C(G)$$

where λ_C is the size of the cut C.

3.1 Useful Properties

Our goal is to compute an ε sparsifier for the graph G we see in the stream. To do this we will need two important properties of sparsifiers, mergeability and composability of cut sparsifiers.

Lemma 13. ***(Mergeability)*** *Let $G_1(V, E_1)$ and $G_2(V, E_2)$ be two graphs such that $E_1 \cap E_2 = \emptyset$. If H_1 and H_2 are ε-sparsifiers of G_1 and G_2 respectively, then $H_1 \cup H_2$ is an ε-sparsifier of $G_1 \cup G_2$.*

Proof. Consider any cut A. Since H_1 is an ε-sparsifier of G_1 we have,

$$(1-\varepsilon)\lambda_A(G_1) \leq \lambda_A(H_1) \leq (1+\varepsilon)\lambda_A(G_1) \tag{1}$$

Since H_2 is an ε-sparsifier of G_2 we have,

$$(1-\varepsilon)\lambda_A(G_2) \leq \lambda_A(H_2) \leq (1+\varepsilon)\lambda_A(G_2) \tag{2}$$

Adding Eq (1) and Eq (2) we get:

$$(1-\varepsilon)\big(\lambda_A(G_1) + \lambda_A(G_2)\big) \leq \lambda_A(H_1) + \lambda_A(H_2) \leq (1+\varepsilon)\big(\lambda_A(G_1) + \lambda_A(G_2)\big)$$

Finally, we use the fact that the graphs do not share any edges i.e. $E_1 \cap E_2 = \emptyset$.

$$\implies \boxed{(1-\varepsilon)\lambda_A(G_1 \cup G_2) \leq \lambda_A(H_1 \cup H_2) \leq (1+\varepsilon)\lambda_A(G_1 \cup G_2)}$$

Thus, $H_1 \cup H_2$ is an ε-sparsifier of $G_1 \cup G_2$. $\square$

Lemma 14. ***(Composability)*** *If H_2 is an ε_2-sparsifier of H_1 and H_1 is a ε_1-sparsifier of G, then H_2 is an $(\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2)$-sparsifier of G.*

Proof. Consider any cut A. Since H_2 is an ε_2-sparsifier of H_1 we have,

$$(1-\varepsilon_2)\lambda_A(H_1) \le \lambda_A(H_2) \le (1+\varepsilon_2)\lambda_A(H_1) \tag{3}$$

Since H_1 is an ε_1-sparsifier of G we have,

$$(1-\varepsilon_1)\lambda_A(G) \le \lambda_A(H_1) \le (1+\varepsilon_1)\lambda_A(G)$$

Thus substituting this in Eq (3) we get,

$$(1-\varepsilon_1)\boxed{(1-\varepsilon_2)\lambda_A(G)} \le \lambda_A(H_2) \le (1+\varepsilon_1)\boxed{(1+\varepsilon_2)\lambda_A(G)}$$
$$(1-\varepsilon_1-\varepsilon_2+\varepsilon_1\varepsilon_2)\lambda_A(G) \le \lambda_A(H_2) \le (1+\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)\lambda_A(G)$$
$$(1-\varepsilon_1-\varepsilon_2-\varepsilon_1\varepsilon_2)\lambda_A(G) \le \lambda_A(H_2) \le (1+\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)\lambda_A(G)$$

Thus H_2 is an $(\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)$-sparsifier of G. □

Note that $(\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2) = (1+\varepsilon_1)(1+\varepsilon_2)-1$. What this is saying is that if H_2 is a $(1+\varepsilon_2)$ approximation of H_1 and if H_1 is a $(1+\varepsilon_1)$ approximation of G then H_2 is a $(1+\varepsilon_1)(1+\varepsilon_2)$ approximation of G.

3.2 Algorithm Idea

A trivial algorithm for getting a sparsifier is to store the whole graph and then create a sparsifier. We could improve this using the properties above. We could divide the stream into two parts. Let G_1 be the graph of the first half of the stream and let G_2 be the graph of the second half of the stream. We could create a sparsifier H_1 for G_1 and then create a sparsifier H_2 for G_2 and then union them to get a sparsifier for H since we know edges do not repeat in the stream. This would still need a lot of space. We have to divide the stream into more than constant number of chunks for an asymptotic improvement. But if the chunk size is too small then maybe the best sparsifier for the graph is the graph itself and we can not get a much smaller sparsifier compared to the graph. We use the following theorem to determine the best chunk size.

Theorem 15. *Every graph with n vertices has a γ-sparsifier with $k = O(\gamma^{-2}n)$ edges.*

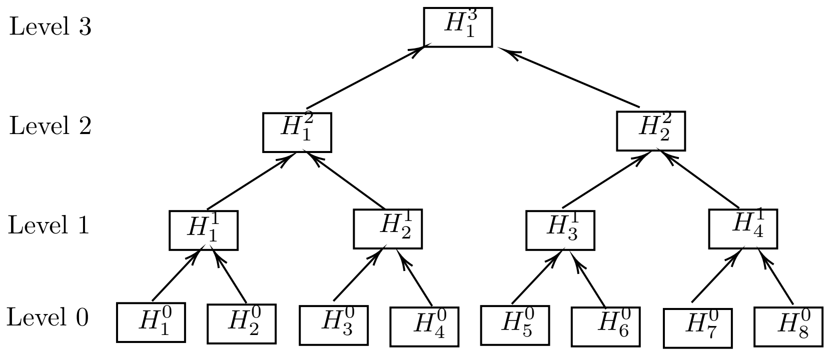
This theorem tells us that no matter how many edges are present in a graph, it always has a weighted γ-sparsifier with at most k edges. Thus, let **ALG** be the algorithm that given any graph outputs a weighted γ-sparsifier with at most k edges.

A bad way of using the properties to create the sparsifier is the following. Divide the stream into t chunks of size k. For each chunk create a sparsifier and merge it with the sparsifier for all the chunks before it. So we start with a sparsifier for the first chuck having k edges. We union it with the sparsifier for the second chuck having k edges and then sparsify the sparsifier from $2k$ edges back to k edges using **ALG**. We maintain one running sparsifier H and for the i^{th} chunk we union the the sparsifier for that chunk with H and reduce the number of edges in H to k using **ALG**. In the end, we have a sparsifier H of the whole stream. This algorithm gives the correct answer in low space but is bad because the approximation factor keeps going up every time you run **ALG**. Thus a better idea would be to combine the sparsifiers in a tree structure to get a better approximation factor.

3.3 Tree structure

In the tree structure below we see that as soon as chunk i (of size k) is seen in the stream, its sparsifier is created which is the graph itself i.e./ all edges in chunk i are in the sparsifier. We ensure that as soon as a level has two sparsifiers, we combine (union and then run **ALG** on the union) them and send it to the next level and then delete them. Note that every sparsifier has at most k edges because for level 0 we consider exactly k edges and for higher levels we run **ALG** on the union of its children which ensures at most k edges.

In general this technique is called merge and reduce. It has lots of applications in streaming. The nice thing about the tree structure is that you do not blow up the error dramatically.



3.4 Algorithm Explanation:

We process the stream in chunks of size k. WLOG we will have $t = \frac{m}{k}$ chunks. For simplicity, assume t is a power of 2 (affects analysis by at most a constant factor). Let G_i be the graph on n vertices with the edges in the i^{th} chunk. For $j \in \{1, \dots, \lg t\}$ and $i \in \{1, \dots, t/2^j\}$, define a weighted subgraph H_i^j recursively as follows:

$$\begin{aligned} H_i^0 &= G_i \\ H_i^j &= \mathsf{ALG}\ \left(H_{2i-1}^{j-1} \cup H_{2i}^{j-1}\right) \quad \text{for } j > 0. \end{aligned}$$

Thus to get the i^{th} sparsifier on level j, we use sparsifier $2i-1$ and sparsifier $2i$ on level $j-1$. We first take their union thus producing a sparsifier with at most $2k$ edges and then run **ALG** on that union getting a sparsifier with at most k edges but a slightly higher approximation factor.

Also, we ensure that as soon as a level has two sparsifiers, we combine (union and then run **ALG** on the union) them and send it to the next level and then delete them. Thus there are at most $\lg t$ number of algorithms **ALG** running in parallel, one at each level.

3.5 Approximation Factor

Lemma 16. *For any i, j in their ranges, H_i^j is a $\big((1+\gamma)^j - 1\big)$-sparsifier of the subtree below it.*

Proof. We will prove this by induction on j (Let i be arbitrary).

Base Case: $j = 0$

At level 0, $H_i^0 = G_i$ thus every cut has the same value in H_i^0 and in G_i. Therefore H_i^0 is a 0 approximation of G_i. Also, when $j = 0$, $(1+\gamma)^j - 1 = 1 - 1 = 0$. Thus the base case is true.

Induction Hypothesis: We assume the lemma is true for $j = l$. In other words, H_i^l is a $\big((1+\gamma)^l - 1\big)$-sparsifier of the subtree below it $(\forall i)$.

Induction Step: We want to prove the lemma for $j = l+1$. Consider any H_i^{l+1}. By definition $H_i^{l+1} = \mathbf{ALG}(H_{2i-1}^l \cup H_{2i}^l)$. We know by Induction Hypothesis that H_{2i-1}^l and H_{2i}^l are $(1+\gamma)^l - 1$ sparsifiers of the subtrees below them. We know by Lemma 13 that H_{2i}^l will be a sparsifier of the subtree below it. Also, by Lemma 14 we know that H_{2i}^l will be an $\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$ sparsifier where $\varepsilon_1 = (1+\gamma)^l - 1$

and $\varepsilon_2 = \gamma$. Thus $\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 = [(1+\gamma)^l - 1] + \gamma + [(1+\gamma)^l - 1]\gamma = (1+\gamma)^{l+1} - 1$. Thus the we proved the lemma for $j = l+1$ and therefore by induction the claim holds for all j. $\square$

It is very easy to see why this is true. At each level the approximation gets worse by a factor of $(1+\gamma)$ because **ALG** is run. Thus at the topmost layer, $\lg t$ the approximation factor is $(1+\gamma)^{\lg t}$ thus $H_1^{\lg t}$ is a $(1+\gamma)^{\lg t} - 1$ approximation of G.

3.6 Parameter tuning

We need to set the value of γ in terms of ε since we need an ε sparsifier. We know that $H_1^{\lg t}$ is a sparsifier of G, the entire graph. Also, $H_1^{\lg t}$ is a $(1+\gamma)^{\lg t}$ approximation of G. We want it to be a $(1+\varepsilon)$ approximation thus $(1+\gamma)^{\lg t} \leq (1+\varepsilon)$. Let $\gamma = \varepsilon/2\lg t$. Thus we have,

$$\begin{aligned}(1+\gamma)^{\lg t} &\leq \exp(\gamma \cdot \lg t) && \text{(Since } 1+x \leq e^x\text{)}\\ &= \exp(\varepsilon/2) && \text{(Substituting } \gamma\text{)}\\ &\leq 1+\varepsilon. && \text{(Since } e^x \leq 1+2x \text{ when } 0 \leq x \leq 1\text{)}\end{aligned}$$

Thus we get an ε sparsifier and the correctness follows.

3.7 Space Analysis:

We observe that as soon as we construct a sparsifier H_i^j, we forget the two other sparsifiers it was created from namely $H_{2i-1}j-1$ and $H_{2i}j-1$. Thus we are never storing more than 2 sparsifiers at each level thus at most $2k$ edges at each level. There are $\lg t$ levels in total thus the total space used is $O(2k \cdot \lg t)$. We know that $t = m/k \leq n^2$ thus $\lg t = O(\log n)$. Also, $k = O(\gamma^{-2}n) = O(n \cdot \varepsilon^{-2}\log^2 t) = O(n \cdot \varepsilon^{-2}\log^2 n)$. Thus the total space used is $O(2k \cdot \lg t) = O(n \cdot \varepsilon^{-2}\log^3 n) = \tilde{O}(n/\varepsilon^2)$.

References

[1] K. J. Ahn and S. Guha. Graph sparsification in the semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 328–338. Springer, 2009. 2

[2] S. Assadi, Y. Chen, and S. Khanna. Polynomial pass lower bounds for graph streaming algorithms, 2019. 2

[3] M. Ghaffari, K. Nowicki, and M. Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1260–1279. SIAM, 2020. 2

[4] M. Henzinger, S. Rao, and D. Wang. Local flow partitioning for faster edge connectivity. *SIAM Journal on Computing*, 49(1):1–36, 2020. 2

[5] A. Rubinstein, T. Schramm, and S. M. Weinberg. Computing exact minimum cuts without knowing the graph, 2019. 2

[6] T. Saranurak. A simple deterministic algorithm for edge connectivity. *arXiv preprint arXiv:2008.08575*, 2020. 2

[7] M. Zelke. Intractability of min-and max-cut in streaming graphs. *Information Processing Letters*, 111(3):145–150, 2011. 2