Towards Building a Scalable Graph Stream Library

Darpana Desai (23110085), Haarit Chavda (23110077), Siddharth Rajandekar (23110310)

Abstract—This report presents the design, implementation, and empirical evaluation of a comprehensive graph streaming library that processes dynamic graph data using sublinear memory constraints. We implemented and analyzed two categories of algorithms: direct graph streaming algorithms (connectivity, minimum spanning trees, spanners, greedy matching, weighted matching, triangle counting, and sparsification) and sketch-based algorithms (connectivity, k-connectivity, min-cut, and sparsification). Our experimental evaluation compared these algorithms on synthetic random graphs while systematically varying graph parameters—increasing nodes with fixed edges, increasing edges with fixed nodes, and scaling both simultaneously—to assess their practical time and space complexity bounds. Results demonstrate that sketch-based approaches generally offer superior theoretical guarantees but incur higher constant factors in practice. We observe that many theoretically efficient algorithms exhibit different practical behaviors than their asymptotic bounds suggest, particularly in the streaming setting. Our findings highlight the gap between theoretical advances and practical implementations in graph streaming, providing valuable insights for deploying these algorithms in real-world applications such as social network analysis, cybersecurity, and dynamic network monitoring. The code for the project is available here - Github Repo

Keywords—*Streaming algorithm, sketch, graph*

1. Introduction

Graphs model complex relationships across domains from social networks to communication systems. As these graphs grow increasingly large and dynamic, traditional algorithms requiring complete graph storage become impractical.

Graph streaming algorithms address this challenge by processing graphs as sequences of edge updates using sublinear memory—typically O(n·polylog n) where n is the number of vertices. This approach enables real-time analytics on massive, evolving networks where storing the complete graph is infeasible.

Despite significant theoretical advances in graph streaming algorithms, practical implementations remain limited. This project implements and evaluates both direct graph streaming algorithms (connectivity, minimum spanning trees, spanners, matching) and sketch-based approaches (connectivity, k-connectivity, min-cut, sparsification) to assess their performance across varying graph parameters.

Our goal is to bridge the theory-practice gap by providing empirical insights into the time and space efficiency of these algorithms on real-world graph structures, identifying which theoretical advances translate effectively to practical applications.

2. Related Work

2.1. On Sampling from Massive Graph Streams

Ahmed et al. introduce Graph Priority Sampling (GPS), a novel reservoir sampling technique for massive graph streams that provides a principled approach to edge sampling while minimizing estimation variance for subgraph counting. Unlike previous methods, GPS decouples the edge sampling process from subgraph estimation, offering two frameworks: Post-Stream estimation for retrospective queries and In-Stream estimation for lower-variance incremental updates. The theoretical foundation of the approach leverages a Martingale formulation of graph stream order sampling, proving that subgraph estimators written as products of constituent edge estimators remain unbiased. Extensive experiments on diverse real-world graphs demonstrate GPS achieving remarkable accuracy (less than 1 percent error for triangle and wedge counting) while storing only a small fraction of edges—notably estimating triangle counts in a Twitter graph with 260M edges at under 1 percent error while storing only 40K edges (0.015 percent of the graph), significantly outperforming previous methods in both accuracy and resource efficiency.

2.2. Graph Stream Algorithms: A Survey

McGregor provides a comprehensive survey of graph streaming algorithms developed over the previous decade, focusing on techniques for processing massive graphs using sublinear space. The paper systematically organizes algorithms by stream type (insert-only, insert-delete, and sliding window models) and problem domain, covering fundamental graph problems including connectivity, spanning trees, sparsification, matchings, and subgraph counting. A central focus is placed on the semi-streaming model, where algorithms are permitted O(n·polylog n) space—a threshold below which many graph problems become provably intractable yet sufficient for practical solutions. The survey highlights key technical innovations such as linear sketching for dynamic graph streams, sparse recovery techniques, and efficient data structures like graph sparsifiers and spanners. For each algorithm class, McGregor presents state-of-the-art complexity results, identifies general algorithmic techniques, and illustrates fundamental ideas through simple examples that illuminate the theoretical foundations while demonstrating practical trade-offs between space complexity, approximation quality, and stream processing constraints.

2.3. Streaming Graph Partitioning for Large Distributed Graphs

Stanton and Kliot address the challenge of partitioning massive graphs across distributed systems by introducing lightweight streaming algorithms that partition graphs as they are loaded into clusters, avoiding the prohibitive computational overhead of traditional offline methods. The authors propose several natural, simple heuristics—primarily variations of greedy approaches—and empirically evaluate them against random hashing (the prevailing approach in distributed systems) and METIS (a fast offline partitioner) across diverse graph datasets. Their comprehensive experimental study reveals significant performance differences between heuristics, with linear weighted deterministic greedy achieving the best performance with an average 76 percent gain over hashing. The approach shows excellent scalability with both graph size and partition count, and when applied to PageRank computation on large social networks (LiveJournal and Twitter), delivered runtime improvements of $18-39$ percent. Most importantly, the research demonstrates that lightweight streaming partitioning can be effectively integrated into existing graph loading processes with minimal overhead, providing a practical solution for distributed graph processing systems where communication costs often become the limiting factor in scaling up computations.

3. Experiments

Table 1. Graph Stream Algorithms Complexity

Algorithm	Time Complexity	Space Complexity
Connectivity (Stream based)	$O(m \cdot \alpha(n))$	$O(n)$
Minimum Spanning Trees	$O(m \cdot \log n)$	$O(n)$
Spanners	$O(mn^{1+1/t})$	$O(n^{1+1/t})$
Greedy Matching	$O(m+n)$	$O(n)$
Greedy Weighted Matching	$O(m+n)$	$O(n)$
Triangle Count	$O(n^3)$	$O(n^3)$
Connectivity (Sketch based)	$O(n \log^3 n)$	$O(n \log^3 n)$
k-Connectivity	$O(kn \cdot \text{polylog } n)$	$O(kn \cdot \text{polylog } n)$
Mincut and Sparsification	$O(n^2 \log n)$	$O(n)$

Streaming Algorithms

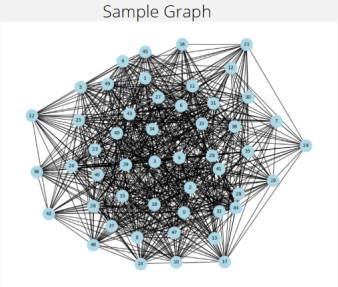

Figure 1. Sample Graph

3.1. Connectivity

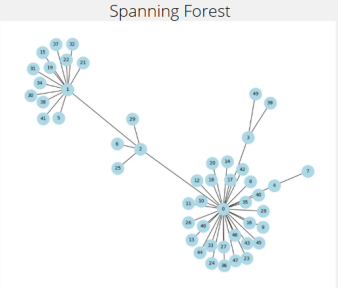

Figure 2. Graph after running Connectivity

Checking Graph Connectivity Using a Single Pass Over the Edge Stream

One of the early motivations for considering the semi-streaming model was that determining whether a graph is connected requires $\tilde{\Theta}(n)$ space, which is both necessary and sufficient. The sufficiency can be demonstrated with a simple algorithm that constructs a spanning forest: we maintain a set of edges H and add the next edge $\{u, v\}$ to H if there is currently no path between u and v in H.

Using the Union-Find Algorithm

Time Complexity: $O(m\alpha(n))$, where m is the number of edges, and $\alpha(n)$ is the inverse Ackermann function, which grows extremely slowly.

Space Complexity:

- H stores at most $n-1$ edges, so space complexity is $O(n)$.
- V_{seen} stores up to n vertices, so space complexity is $O(n)$.

3.2. Minimum Spanning Trees

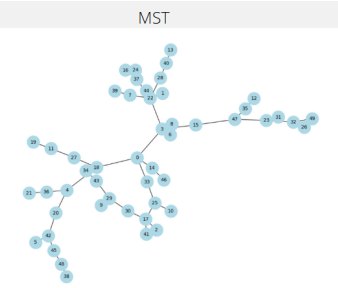

Figure 3. Graph after running MST

What is an MST

A **Minimum Spanning Tree (MST)** is a subgraph of a connected, weighted graph that connects all the nodes together with the minimum possible total edge weight and no cycles. If the graph is not connected, we construct a **Minimum Spanning Forest** (a collection of MSTs, one for each connected component).

Streaming Model

In the streaming model, edges arrive one by one in a stream. We do not have access to the entire graph in advance, and we aim to process each edge quickly and with limited memory, making standard MST algorithms unsuitable without adaptation.

Time Complexity:

- Per edge: $O(\alpha(n))$ using Union-Find with path compression
- Total (for m edges): $O(m \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function (very slow-growing)

Space Complexity:

- $O(n)$ for Union-Find parent and rank arrays
- $O(n)$ for storing the MST edges (since an MST has at most $n-1$ edges)

3.3. Spanner

Explanation of the Algorithm

A **spanner** is a sparse subgraph that approximately preserves distances between all pairs of nodes in the original graph. Specifically, a subgraph H is called an α-spanner of graph G if, for all nodes $u, v \in V$:

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

where d_G and d_H represent the shortest path distances in G and H, respectively.

The spanner construction algorithm works by processing edges from a stream S and deciding whether to add each edge based on whether it significantly shortens any path. Unlike connectivity algorithms that avoid forming cycles, this algorithm adds an edge if it doesn't complete a short cycle—specifically if the current path between two nodes is longer than α times the shortest path in G.

Time and Space Complexity

Space Complexity:

- Nodes: $O(n)$
- Edges: $O(n^{1+1/t})$
  When $\alpha = 2t - 1$ for some integer t, the number of edges stored in the spanner is at most $O(n^{1+1/t})$.

Time Complexity:

- Per edge: $O(n + n^{1+1/t})$
  (Includes shortest path query + insertion check)
- Total: $O(m \cdot n^{1+1/t})$
  Where m is the total number of edges in the input stream.

3.4. Matchings

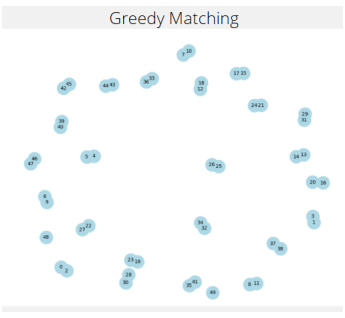

Figure 4. Unweighted Greedy Matching of the Graph

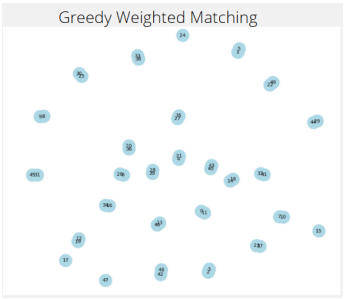

Figure 5. Weighted Greedy Matching of the Graph

Explanation

A **matching** in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in M share an endpoint.

In the **streaming model**, we cannot store the entire graph. Instead, edges arrive one-by-one, and we must make decisions with limited memory and in a single pass.

Unweighted Matching (Greedy Matching Algorithm)

A simple greedy algorithm gives a 2-approximation to the maximum matching.

Intuition: When an edge arrives, include it in the matching if and only if it does not share endpoints with any edge already in the matching.

Weighted Matching (Greedy with Threshold Rule)

For the weighted case, the greedy strategy is modified to include a threshold condition using a parameter $\gamma > 0$.

Key Idea: Only add a new edge if its weight is sufficiently larger than the weight of conflicting edges, ensuring that low-weight "trails" of added and later removed edges do not dominate the total weight.

Time and Space Complexity

Unweighted Greedy Matching

Time Complexity (per edge): $O(1)$ for constant-time endpoint checks (with proper data structures)

Total Time Complexity: $O(m)$, where m is the number of edges

Space Complexity: $O(n)$ to store the matching M, since a matching has at most $n/2$ edges

Weighted Greedy Matching

Time Complexity (per edge): $O(1)$ to identify conflicts and compute weight comparisons (assuming fast data structures)

Total Time Complexity: $O(m)$ again assuming efficient implementation

Space Complexity: $O(n)$ to store matching and associated weights

3.5. Triangle Counts

Sample Graph:
50 nodes and 720 edges

Stream spectral sparsifier:
Epsilon = 0.2
Edges in original graph: 720
Edges in sparsified graph: 718

Using l_0 sampling:
Estimated triangles: 3851
True number of triangles: 3950

Figure 6. Triangle Counts for the Graph

Problem Overview

A **triangle** in a graph is a set of three nodes that are mutually connected. Triangle counting is fundamental for analyzing:

- **Transitivity:** Fraction of 2-length paths that close into triangles.
- **Clustering coefficient:** Measures how "tightly-knit" a network is.

Let T_3 be the total number of triangles in a graph $G = (V, E)$.

Challenge in Streaming Model

Exact triangle counting requires $\Omega(n^2)$ space — infeasible for large graphs.

Any constant-factor approximation requires $\Omega(m/T_3)$ space.

Goal: Estimate T_3 using sublinear space and one pass over the edge stream.

Algorithmic Approaches

A. Vector-Based Frequency Moment Approach

Theory: Define a vector x indexed by all 3-node subsets $T \subseteq V$,

$$x_T = \#\text{edges in } T$$

Then,

$$T_3 = \#\{T : x_T = 3\}$$

We use the following identity (Lemma 2.1):

$$T_3 = F_0 - 1.5F_1 + 0.5F_2$$

Where:

- $F_0 = \#\text{non-zero } x_T$
- $F_1 = \sum_T x_T$
- $F_2 = \sum_T x_T^2$

Complexity:

- **Space:** $\tilde{O}\left(\frac{1}{\varepsilon^2} \cdot \left(\frac{m}{nt}\right)^2\right)$ for a $(1 \pm \varepsilon)$-approximation, assuming $t \leq T_3$ is a lower bound.
- **Time per edge:** $O(1)$ for moment sketches.
- **Passes:** One pass (streaming).

B. ℓ_0 Sampling Approach

Theory: Randomly sample a triple T where $x_T \neq 0$ using ℓ_0-sampling.

Set $Y = 1$ if $x_T = 3$, else $Y = 0$.

Since $\mathbb{E}[Y] = T_3/F_0$, we can estimate T_3 by:

$$\hat{T}_3 = \hat{F}_0 \cdot \text{mean of multiple } Y$$

Complexity:

- **Space:** $\tilde{O}\left(\frac{1}{\varepsilon^2} \cdot \frac{m}{nt}\right)$
- **Time per edge:** $O(\text{polylog } n)$
- **Passes:** One pass

3.6. Sparsifier

Problem Overview

Graph sparsification aims to construct a subgraph H of a graph G that approximately preserves specific connectivity properties like cut values or spectral properties. These sparsifiers are useful for efficiently approximating graph metrics like cuts and random walks, which are fundamental in various applications including network analysis and machine learning.

Cut Sparsification

A weighted subgraph H is a $(1 + \varepsilon)$ cut sparsifier of graph G if for every subset $A \subset V$, the weight of the cut in G and H is approximately the same:

$$\lambda_A(H) = (1 \pm \varepsilon)\lambda_A(G)$$

Where:

$$\lambda_A(G) \text{ and } \lambda_A(H)$$

represent the weights of the cut $(A, V \setminus A)$ in graphs G and H, respectively.

Spectral Sparsification

A weighted subgraph H is a $(1+\varepsilon)$ spectral sparsifier of graph G if for every vector $x \in \mathbb{R}^n$:

$$x^T L_H x = (1 \pm \varepsilon) x^T L_G x$$

Where L_G and L_H are the Laplacians of graphs G and H, respectively. This condition ensures that H approximates the spectral properties of G, including the eigenvalues and effective resistances of the graph.

Constructing Spectral Sparsifiers

Segmenting the Stream: The input stream of edges is partitioned into $t = \frac{m}{\text{size}(\gamma)}$ segments, each of size size(γ). Here, γ is a parameter that controls the approximation accuracy.

Constructing Local Sparsifiers: For each segment, we apply any existing algorithm (denoted A) that returns a $(1+\gamma)$ spectral sparsifier. The algorithm A is assumed to produce sparsifiers with $O(\gamma^{-2}n)$ edges.

Merging Sparsifiers: After processing each segment, the local sparsifiers are merged and reduced recursively:

- First, merge two sparsifiers into a new one.
- Then, reduce the resulting sparsifiers further by recursively merging them.

This process ensures that the final sparsifier approximates the spectral properties of the entire graph.

Final Sparsifier: The final sparsifier H after $\log_2 t$ recursive merges will be a $(1+\varepsilon)$ spectral sparsifier.

Properties:

- **Mergeable:** If H_1 and H_2 are spectral sparsifiers for G_1 and G_2, then $H_1 \cup H_2$ is a spectral sparsifier for $G_1 \cup G_2$.
- **Composable:** If H_3 is a spectral sparsifier for H_2 and H_2 is for H_1, then H_3 is a spectral sparsifier for H_1.

Space Complexity Analysis

Space per Segment: For each segment, we store $O(\gamma^{-2}n)$ edges as sparsifiers.

Total Space: After $\log_2 t$ merges, the space required for storing the sparsifier is:

$$O(\gamma^{-2} n \log_2 t) = O(\varepsilon^{-2} n \log^3 n)$$

Thus, the space complexity is sublinear and efficiently scales with the graph size, making this approach suitable for streaming algorithms.

Time Complexity

Time per Segment: For each edge in the segment, the algorithm processes it in constant time $O(1)$.

Total Time: The total time complexity is $O(m \cdot \gamma^{-2} n \log_2 t)$, where m is the number of edges in the graph.

Sketching Algorithms

3.7. Connectivity

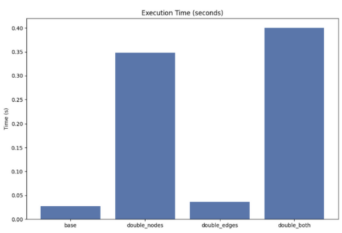

Figure 7. Time Complexity of Connectivity Algorithm

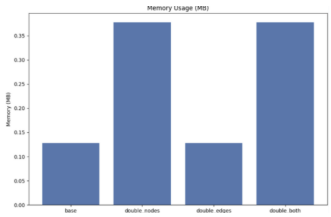

Figure 8. Space Complexity of Connectivity Algorithm

Overview

We study connectivity in large graphs using sketching to reduce memory usage. The goal is to find connected components efficiently in a streaming setting.

Vector Encoding

Each node $v_i \in V$ is assigned a vector $a_i \in \{-1, 0, 1\}^{2n}$ with non-zero values indicating participation in an edge:

- $+1$ if v_i is the first node in an edge,
- -1 if it's the second node,
- 0 otherwise.

These vectors capture edge information for sketch-based processing.

ℓ_0-Sampling

We use random projections $M_r \in \mathbb{R}^{k \times d}$ (with $k = O(\log^2 n)$) to sample non-zero entries in a_i. This allows edge recovery using linear sketches.

Connectivity via Sketches

Step 1: Sketching. Pick $t = O(\log n)$ random seeds. For each, compute $M_r a_i$ for all nodes.

Step 2: Forest Construction. Simulate a spanning forest:

- Start with each node as a supernode.
- In each round, sample one outgoing edge per supernode.
- Merge connected supernodes.

Repeat for $O(\log n)$ rounds. The graph is connected if one supernode remains.

Complexity

Time:

$$O(m \log^2 n + n \cdot \text{polylog}(n)) \quad (\text{or } O(n \log^2 n) \text{ for sparse graphs})$$

Space:

$$O(n \cdot \text{polylog}(n)) \quad \text{for all sketches}$$

3.8. K-Connectivity

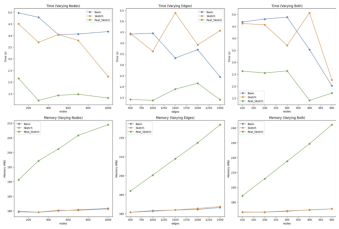

Figure 9. k-Connectivity

Overview

A graph is k-connected if every cut has at least k edges, ensuring robustness under up to $k-1$ edge removals. We extend sketch-based connectivity methods to test k-connectivity efficiently.

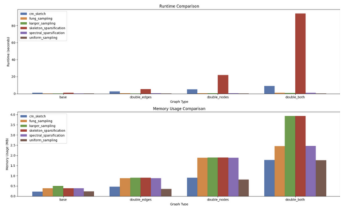

Figure 10. Mincut and Sparsification

Sketch-Based Algorithm

The key idea is:

$$A(G - F) = A(G) - A(F)$$

We use this to build k spanning forests incrementally by removing edges from previous forests.

Steps:

- Compute k independent sketches $A_1(G), \ldots, A_k(G)$.
- For each i, compute F_i on $G \setminus \{F_1, \ldots, F_{i-1}\}$ using:

$$A_i(G - \cup_{j=1}^{i-1} F_j) = A_i(G) - \sum_{j=1}^{i-1} A_i(F_j)$$

Complexity

Time:

$O(km \log^2 n + kn \cdot \text{polylog}(n))$ or $O(kn \cdot \text{polylog}(n))$ for sparse graphs

Space:

$$O(kn \cdot \text{polylog}(n))$$

3.9. Mincut and Sparsification

Sparsification via Sampling

The results presented here build on a generic sampling algorithm, which works as follows:

Generic Sparsification Algorithm:

- **Sampling:** Sample each edge e with probability p_e.
- **Edge Weighting:** Weight each sampled edge e by $\frac{1}{p_e}$.

It is straightforward to see that the size of any cut in the graph is preserved in expectation through this process. When the probability p_e is sufficiently large, various graph properties—including the size of cuts—are approximately preserved with high probability. Specifically, for a constant c_1, if

$$p_e \geq q := \min\left(1, c_1 \lambda^{-1} \epsilon^{-2} \log n\right),$$

where λ is the size of the minimum cut in the graph, the resulting graph will be a cut sparsifier with high probability.

Minimum Cut

As a warm-up, we present an algorithm for estimating the minimum cut λ of a dynamic graph, using Karger's sampling result in conjunction with the skeleton construction algorithm from the previous section. For every graph G_i formed by subsampling edges of G (with each edge included with probability $\frac{1}{2^i}$), we construct a k-skeleton $H_i = \text{skeleton}_k(G_i)$ for $k = 3c_1\epsilon^{-2} \log n$.

Let $j = \min\{i : \text{mincut}(H_i) < k\}$. We claim that:

$$2j \cdot \text{mincut}(H_j) = (1 \pm \epsilon)\lambda.$$

For $i \leq \log_2(1/q)$, Karger's result ensures that all cuts are approximately preserved. In particular,

$$2i \cdot \text{mincut}(H_i) = (1 \pm \epsilon)\text{mincut}(G_i).$$

When $i = \log_2(1/q)$, the expected minimum cut $\text{mincut}(H_i)$ satisfies:

$$E[\text{mincut}(H_i)] \leq 2^{-i}\lambda \leq 2q\lambda \leq 2c_1\epsilon^{-2} \log n,$$

and by applying the Chernoff bound, we find that $\text{mincut}(H_i) < k$ with high probability. Thus, $j \leq \log_2(1/q)$ with high probability, and Equation (4) holds.

3.10. Sparsification

To construct a sparsifier, we sample edges with probability

$$p_e = \min\left(1, \frac{\lambda_e}{t}\right),$$

for some value t. If $t = \Theta(\epsilon^{-2} \log^2 n)$, then the resulting graph is a combinatorial sparsifier. If $t = \Theta(\epsilon^{-2} n^{2/3} \log n)$, the graph is a spectral sparsifier.

The main challenge in this process is that we do not know the values of λ_e in advance. To overcome this, we adopt a strategy similar to the one used in estimating the minimum cut. Specifically, let G_i be defined as above, and let $H_i = \text{skeleton}_{3t}(G_i)$. Assuming $\lambda_e \geq t$, we claim that:

$$\Pr[e \in H_0 \cup H_1 \cup \cdots \cup H_{2\log n}] \geq \frac{\lambda_e}{t}.$$

This follows because the probability is at least $\Pr[e \in H_j]$ for $j = \log\left(\frac{\lambda_e}{t}\right)$. The expected size of the minimum cut separating edge $e = \{u, v\}$ in H_j is at most $2t$, and applying the Chernoff bound ensures that this cut is at most $3t$ with high probability. Hence, the probability $\Pr[e \in H_j]$ is approximately equal to $\Pr[e \in G_j]$, since H_j is a $3t$-skeleton.

Thus, we conclude that the sparsifier is formed with high probability, as

$$\Pr[e \in G_j] \geq \frac{\lambda_e}{t}.$$

4. Results

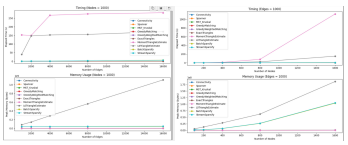

Figure 11. Performance of streaming algorithms on the given datasets.

This is the performance of the various algorithms as tested on multiple datasets. For the plots on the left-hand side, we have kept the nodes constant and are varying the number of edges. On the right, we have kept the number of edges constant, and the nodes vary. In the top plots, we compare the time taken for the algorithms to run, and in the bottom plots, we compare the memory taken by the algorithms.

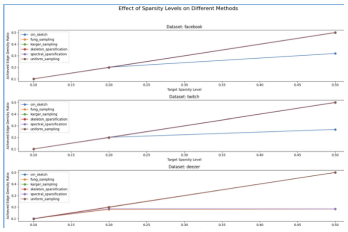

Figure 12. Effect of Sparsification on different methods

This plot shows the comparison between sparsity level vs edge density ratio. We have three plots - one for each of the datasets (facebook, twitch and deezer). The plot tells us how is the edge density level of the graph affected when we vary its sparsity.

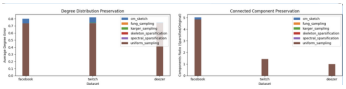

Figure 13. Degree Distribution and Connected Component preservation in the sparsified graph

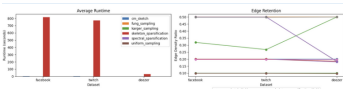

Figure 14. Average Runtime and edge retention in the sparsified graph.

5. Future Work

- Find more streaming or sketching algorithms, like count min sketch, sliding window, etc. and implement them in Python.
- Making a fully fledged library that would allow the user to directly use the implemented sketching and streaming algorithms.
- Making an AI chatbot specifically for graph algorithms.
- Interactive tool that would address these graph-based queries.