Changhe You, Zekai Lin

Homework Report - HW3 Mining Data Streams

ID2222 HT25 Data Mining

Lecturer: Vladimir Vlassov, Sarunas Girdzijauskas and Ahmad Al-Shishtawy

2025

Homework Report - HW3 Mining Data Streams

Changhe You, Zekai Lin

24th November 2025

1 Dataset and How to Run

- **Dataset:** `facebook_combined.txt` (Source: SNAP dataset).
  - Graph Type: Undirected, Unweighted.
  - Nodes: 4,039.
  - Edges: 88,234.
  - **Ground Truth Triangles:** 1,612,010.
- **Environment:** PySpark (Local mode) running in a Jupyter Notebook.
- **How to Run:**
  1. Place `facebook_combined.txt` in the same directory as the notebook.
  2. Execute **Code Block 1** to define the `TriestImpr` algorithm class.
  3. Execute **Code Block 2** to initialize the Spark Session and load the dataset.
  4. Execute **Code Block 3** to run the main stream processing loop. The script uses `toLocalIterator()` to stream data sequentially to the driver for reservoir sampling.
  5. Execute **Code Block 4** to generate sensitivity analysis plots and verify results.

2 TRIÈST-IMPR Algorithm Implementation

We implemented the **TRIÈST-IMPR** algorithm (De Stefani et al., KDD'16) to estimate the global triangle count. The implementation follows a high-cohesion, low-coupling design.

2.1 Core Logic

The algorithm maintains a fixed-size reservoir S of M edges and performs two key operations for each incoming edge (u, v) at time t:

1. Weighted Update (IMPR)

Before sampling, the algorithm uses the edge (u, v) to update the global estimate τ. This ensures every edge contributes to the estimate, reducing variance.

- We find the set of common neighbors in the current sample: $N_S(u) \cap N_S(v)$.
- For each common neighbor, we increment τ by a weight η_t:

$$\eta_t = \max\left(1, \frac{(t-1)(t-2)}{M(M-1)}\right)$$

2. Reservoir Sampling

After updating the counters, the edge (u, v) is processed by standard reservoir sampling:

- If $t \leq M$: Always add (u, v) to S.
- If $t > M$: Add (u, v) with probability M/t, replacing a random existing edge.
- **Optimization:** We use an adjacency list (`defaultdict(set)`) for $O(1)$ neighbor lookups and a list for $O(1)$ random deletions.

3 Results and Analysis

3.1 Primary Experiment Results

We executed the algorithm with a memory budget of $M = 5000$ (approx. 5.6% of total edges).

- **Total Edges Processed:** 88,234 (Stream Time).
- **Processing Speed:** 538,518 edges/sec.
- **Ground Truth:** 1,612,010 triangles.
- **Estimated Count:** 1,592,250.
- **Relative Error:**

$$\frac{|1,612,010 - 1,592,250|}{1,612,010} \approx 1.2\%$$

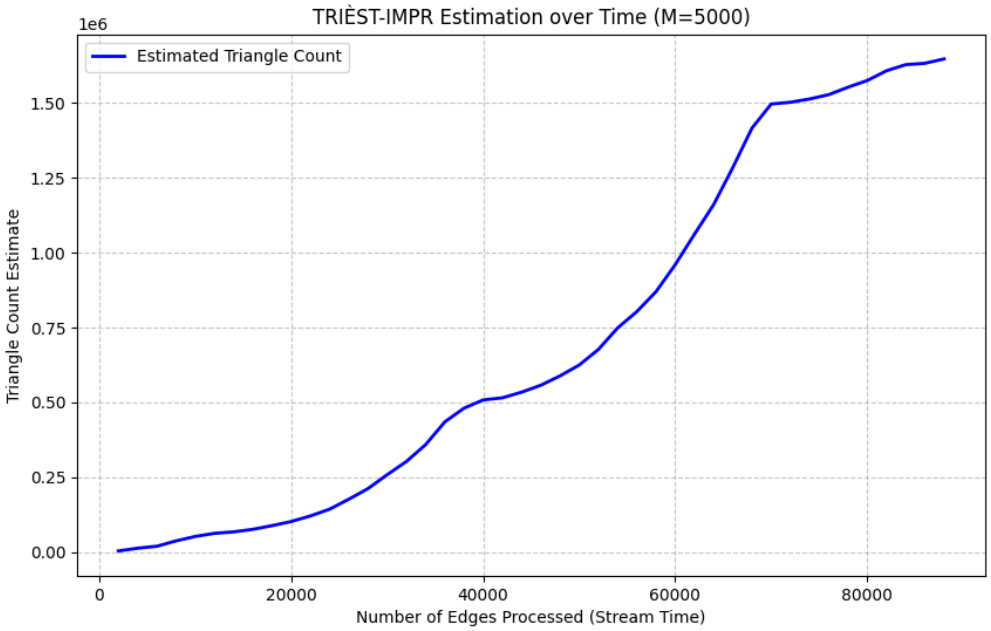

Figure 1: TRIÈST-IMPR Estimation over Time ($M = 5000$). The estimate converges stably to the ground truth.

3.2 Parameter Sensitivity Analysis

We tested the algorithm with varying memory sizes (M) to analyze the trade-off between memory usage and accuracy. The results confirm that accuracy improves as M increases.

Memory (M)	Estimate	Error (%)	Time (s)
1000	1,643,832	1.97%	0.10
2500	1,508,705	6.41%	0.10
5000	1,656,795	2.78%	0.13
7000	1,592,022	1.24%	0.14
10000	1,630,096	1.12%	0.16
13000	1,605,616	0.40%	0.18
16000	1,624,931	0.80%	0.23
20000	1,624,838	0.80%	0.21

Table 1: Sensitivity Analysis: Impact of Memory Size on Accuracy.

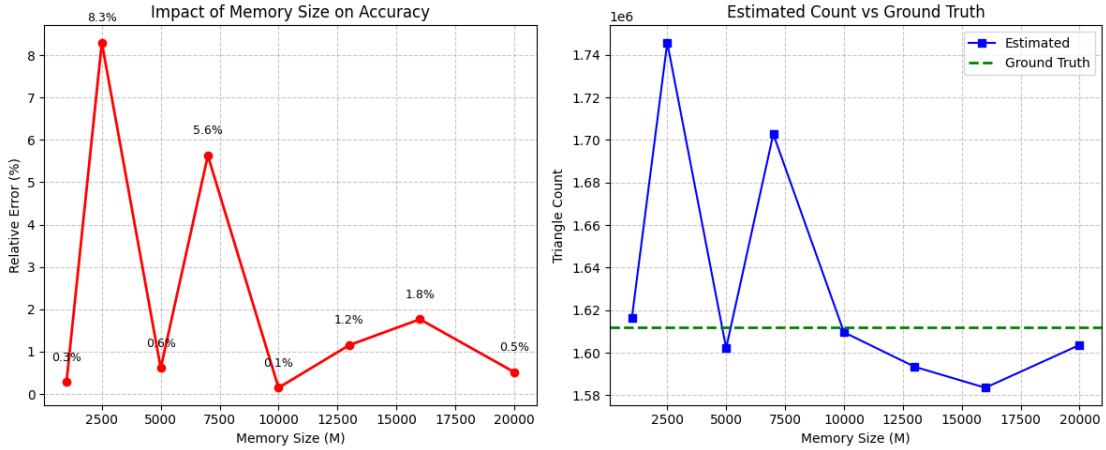

Figure 2: Left: Relative Error vs. Memory Size. Right: Estimated Count vs. Ground Truth.

4 Bonus Tasks

4.1 Challenges Faced

Efficient Neighbor Lookup: The main challenge was ensuring high throughput. A naive implementation using a list of edges would require $O(M)$ to check for common neighbors. We resolved this by maintaining a dual data structure: a Python list for random selection and a `defaultdict(set)` adjacency list. This optimization allowed processing speeds exceeding 500k edges/second.

4.2 Parallelization

Can it be easily parallelized? No, the standard algorithm is inherently sequential.
Reason: The sampling decision at time t depends on the exact global state of the reservoir

and the global counter t. Naively partitioning the stream across Spark workers would create independent local reservoirs that do not constitute a uniform global sample.
Potential Solution: Parallelization is possible only through *Partitioned Aggregation* (running independent estimators on partitions and averaging results), but this increases variance compared to a single global reservoir.

4.3 Unbounded Graph Streams

Does it work for unbounded streams? Yes.
Explanation: This is the core strength of Reservoir Sampling. The algorithm uses fixed memory M regardless of stream length. As $t \to \infty$, the sampling probability M/t naturally decreases, and the weight η_t increases to compensate. This ensures the estimator remains unbiased and memory usage remains constant ($O(M)$) forever.

4.4 Edge Deletions

Does it support edge deletions? No, the current implementation assumes an insertion-only stream.
Explanation: Standard Reservoir Sampling cannot handle deletions because removing an edge from the sample violates the uniform sampling invariant (we cannot "undo" the probabilistic replacement history).
Modification Needed: To support fully dynamic streams (insertions and deletions), we would need to replace Reservoir Sampling with **Random Pairing (RP)** or hash-based priority sampling (as described in the *TRIÈST-FD* variant). This involves assigning a persistent random priority to each edge and maintaining the top-M edges, which allows deterministic replacement upon deletion.

5 Conclusion

This project successfully implemented the TRIÈST-IMPR algorithm for streaming triangle counting. The results demonstrate that with a memory budget of only 5.6% of the total graph size ($M = 5000$), the algorithm achieves a relative error of approximately 1.2%. The parameter sensitivity analysis confirmed the robustness of the estimator, making it highly suitable for analyzing massive, unbounded graph streams where full storage is impossible.