Report 3: Mining Data Streams

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 24, 2025

1 Introduction

In this homework we implemented an algorithm described in a paper on streaming graph processing. We chose the paper *TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size* [1].

The paper presents one-pass streaming algorithms that compute unbiased, high-quality approximations of the global and local (per-vertex) numbers of triangles in a fully dynamic graph, where updates are adversarial streams of edge insertions and deletions.

We implemented Triest-Base and Triest-FD, two of the algorithms described in the paper. Triest-Base does not support fully-dynamic streams, while Triest-FD is the version that handles both insertions and deletions. Both algorithms rely on reservoir sampling to randomly select a sample (without replacement) of k items from a population of unknown size n (a data stream).

2 Our Methods

An important decision was selecting the data structure to store stream edges read from the input file. We considered using a set of tuples or a set of frozensets. It was not obvious which option would be more efficient, but we chose frozensets because the algorithm targets undirected graphs: with no direction, storing a tuple for each orientation would require two entries per edge, which is inefficient in memory. In addition, since the Base and Improved algorithms do not support edge deletion, using a set of frozensets is a natural choice.

We also keep track of neighbors for each vertex using a *defaultdict[int, set[int]]*. This structure is essential because it speeds up processing of incoming edges from the data stream.

[1]de Stefani, Epasto, Riondato, Upfal (2016). Available at: `https://www.kdd.org/kdd2016/papers/files/rfp0465-de-stefaniA.pdf?courseID=57474&assignmentID=351128&skipModuleItemSequence=true`

2.1 Reservoir sampling

This technique is used when selecting whether to include an incoming edge in the sample or discard it and keep the previous sample. This is done in the following function and uses the Bernoulli random variable and the random.choice() method.

```
def _sample_edge(self, e: Edge) -> tuple[bool, Edge | None]:
    if self.t <= self.M:
        return True, None
    accept_prob = self.M / float(self.t)
    # Bernoulli trial to decide whether to keep or discard
    if bernoulli.rvs(p=accept_prob):
        if not self.S:
            return True, None
        evict = random.choice(tuple(self.S))
        return True, evict
    return False, None
```

This function returns a tuple consisting of a boolean (True if the edge should be included) and the Edge to be discarded from the sample (or None if the new edge is not added).

2.2 TriestBase

Below is the pseudocode for this algorithm, which serves as the basis for Triest-Improved.

```
Algorithm 1 TRIÈST-BASE
Input: Insertion-only edge stream $\Sigma$, integer $M \geq 6$
$\mathcal{S} \leftarrow \emptyset$, $t \leftarrow 0$, $\tau \leftarrow 0$
for each element $(+, (u, v))$ from $\Sigma$ do
    $t \leftarrow t + 1$
    if SampleEdge($(u, v), t$) then
        $\mathcal{S} \leftarrow \mathcal{S} \cup \{(u, v)\}$
        UpdateCounters($+, (u, v)$)

function SampleEdge($(u, v), t$)
    if $t \leq M$ then
        return True
    else if FlipBiasedCoin($\frac{M}{t}$) = heads then
        $(u', v') \leftarrow$ random edge from $\mathcal{S}$
        $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(u', v')\}$
        UpdateCounters($-, (u', v')$)
        return True
    return False

function UpdateCounters($(\bullet, (u, v))$)
    $\mathcal{N}^{\mathcal{S}}_{u,v} \leftarrow \mathcal{N}^{\mathcal{S}}_u \cap \mathcal{N}^{\mathcal{S}}_v$
    for all $c \in \mathcal{N}^{\mathcal{S}}_{u,v}$ do
        $\tau \leftarrow \tau \bullet 1$
        $\tau_c \leftarrow \tau_c \bullet 1$
        $\tau_u \leftarrow \tau_u \bullet 1$
        $\tau_v \leftarrow \tau_v \bullet 1$
```

Figure 1: Illustration of the TRIÈST Base algorithm.

The global counter is t and the local counters (τ) are stored as *defaultdict[int, float]*.

The first part of the pseudocode is implemented in the function **run**, which inserts all edges read from the input file.

I also added the mathematical constant *xi* as a *@property* of the class, defined as

```
 @property
def xi(self) -> float:
    return max(
        1.0,
        self.t
        * (self.t - 1)
        * (self.t - 2)
        / (self.M * (self.M - 1) * (self.M - 2)),
    )
```

The function `run()` returns the estimated number of global triangles, computed as

```
return self.xi * self.tau
```

I also describe the function **_update_counters**, which is common to both the base and improved implementations.

```
def _update_counters(self, e: Edge, inc: float):
        if len(self.S) == 0:
            return

        u, v = tuple(e)
        common = self._common_neighbours(e)
        if not common:
            return

        for c in common:
            self.tau += inc
            self.tau_vertices[u] += inc
            self.tau_vertices[v] += inc
```

As shown, we update the global and local triangle counters. The only nontrivial part is extbf_common_neighbours, which computes the common neighborhood of the two nodes of an inserted edge.

2.3 Triest-Improved

Triest-Improved inherits many methods from the base class, although some behaviors differ. First, the global estimate is computed as

```
return float(self.tau)
```

We also compute another constant called *eta*, as shown below.

```
return max(1.0, (self.t - 1) * (self.t - 2) / (self.M * (self.M - 1)))
```

The main differences in processing each incoming edge are:

- **UpdateCounters** is called unconditionally for each stream element, before the algorithm decides whether to insert the edge into S.
- Triest-Improved never decrements the counters when an edge is removed from S.
- **UpdateCounters** performs a weighted increase of the counters using *eta* as the weight.

2.4 Testing

We developed tests for both the base and improved algorithms. Using a memory size of $M = 10000$ (this value can be changed), the estimates are close to the true value reported for the dataset (error less than 5%). However, there is noticeable variance between estimates, especially for the base algorithm. In the improved-algorithm test we include a plot showing estimates obtained for different values of M (fractions or multiples of the initial M).

Regarding runtime, the algorithm is fast in our implementation: it finishes in less than two seconds for the tested parameters. In an initial implementation that did not store node neighborhoods, the runtime was about 20 seconds. This demonstrates that the chosen data structure for neighbors significantly improves performance.

3 Extra questions

What were the challenges you faced when implementing the algorithm? The paper explains the algorithm clearly, especially via pseudocode, and the procedure was mostly straightforward. The primary challenge was choosing appropriate data structures to store the graph edges; we discussed and motivated our choice in the Methods section.

Can the algorithm be easily parallelized? Introducing data parallelism is not straightforward because the data arrives as a stream and updates to the counters depend on the current state of the sample set, which can be modified dynamically. These dependencies make simple parallelization challenging.

Does the algorithm work for unbounded graph streams? Yes. The algorithm is designed for data streams (which are, by definition, unbounded). It can be queried at any time to provide the current estimate of the number of global triangles. The algorithm only needs the sample set, a triangle counter, and the number of processed stream elements to compute the estimate.

Does the algorithm support edge deletions? The implementations we developed (Triest-Base and Triest-Improved) do not support edge deletions; this is a limitation. The paper also describes *Triest-FD*, an extension that handles fully-dynamic streams with both insertions and deletions. Triest-FD uses Random Pairing (RP), keeping track of edges deleted from the sample and the total number of deletions. This information affects whether new edges are inserted into the sample set and yields an improved estimation formula for the number of triangles.