Report 3: Mining Data Streams

Group 150: Lorenzo Deflorian, Riccardo Fragale

November 23, 2025

1 Introduction

In this homework we were asked to implement the algorithm defined in a paper related to a stream graph processing algorithm. We decided to select the paper called *TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size* [1] written by de Stefani, Epasto, Riondato and Upfal in 2016. It defines a set of one-pass streaming algorithms to compute unbiased, high-quality approximations of the global and local (i.e., incident to each vertex) number of triangles in a fully-dynamic graph represented as an adversarial stream of edge insertions and deletions. We decided to implement only Triest-Base and Triest-FD, the first two algorithms presented in the paper which does not support a fully-dinamic sream as input. Both algorithms use the reservoir sampling technique to randomically choose a simple random sample, without replacement, of k items from a population of unknown size n (actually a data stream).

2 Our Methods

A first important thing to notice is the selection of the data structures where to store all the edges of the stream contained in the input file. The choice for us seemed to be between a set of tuples and a set of frozensets. It was not easy to define which of the two was better in terms of speed and efficiency but we ended up choosing the frozensets solution.

Since this algorithm is explicitly meant for undirected graphs there was no real reason in using tuples as without the direction we would have to store two tuples for each edge (definitely not the best in terms of memory consumption). Considering also that both the Base and the Improved algorithm doesn't support edge deletion, using a set of frozensets was the best choice in our opinion.

[1] Paper: *TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size*, 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`

We also decided to keep track of the neighbours for each edge. This is maintained through a *defaultdict[int, set[int]]*. It is essential since it helps us to speed up the process when processing new esges of the graph contained in the data stream.

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 is returning a Tuple made of a boolean (True if the edge has to be included) and the Edge to be discarded from the sample (None in case the new Edge is not added to the sample).

2.2 TriestBase

I will add below the pseudocode for this algorithm which is also a base for TriestImproved.

Algorithm 1 TRIÈST-BASE

Input: Insertion-only edge stream Σ, integer $M \geq 6$

$\mathcal{S} \leftarrow \emptyset$, $t \leftarrow 0$, $\tau \leftarrow 0$
for each element $(+, (u, v))$ from Σ **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 gloabl counter is t and the local counters tau are stored as *default-dict[int, float]*.

The first part of the pseudocode is implemented inside the function **run** and simply includes the insertion of all the edges mined from the input file.

I also added as *@property* of the class the mathematical constant xi 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 counter of gloabl triangles computed as

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

I will also describe here the function **_update_counters** which is common for the base and improved implementation.

```
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 it can be easily seen we are simply modifying and updating the counters for global and local triangle. The only interesting thing is **_common_neighbours** to which calculates the common neighbourhood between the two nodes of an inserted edge.

2.3 Triest-Improved

The class Triest-Improved inherits many methods from the basic class even though some things are a bit different. First of all the global estimated is computed as

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

We also compute another constant called *eta* computed as shown in the code snippet below.

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

The other main differences are related to the processing of each incoming edge. We have 3 main differences;

- UpdateCounters is called unconditionally for each element on the stream, before the algorithm decides whether or not to insert the edge into S
- trièst-impr never decrements the counters when an edge is removed from S
- UpdateCounters performs a weighted increase of the counters using *eta* as weight

2.4 Testing

We developed a test both for the basic and the improved algorithm. In both cases the estimate is not very different from the real value contained in the description of the dataset. Looking at the amount of time needed to

produce the result we can easily say that the algorithm is very fast since it finishes in less than two seconds. In one of our initial implementation, that didn't include the storage of neighbourhoods of nodes, the time needed was 20 seconds. This implies that we definitely needed that data structures improvement.

3 Extra questions

What were the challenges you faced when implementing the algorithm? The algorithm was explained very well, especially in terms of pseudocode. Moreover, the procedure was quite straightforward and there were only minor differences between the Base and the Improved version. We found a bit hard to correctly identify the data structures to correctly store the set of edges of the graph. We describe our choice in the previous section related to the methods we used.

Can the algorithm be easily parallelized? We believe that introducing data parallelism in the algorithm doesn't really make a sense since the data is flowing into the algorithm through a data stream and that the operations on the counters are blocking. Moreover, the operations on the counters rely on the current state of the sampleset, which can be modified dynamically.

Does the algorithm work for unbounded graph streams? The algorithm works on data streams, that are by definition unbounded. In addition, it can be queried anytime and it gives you the current estimation for the number of global triangles. As visible in the experiments, the algorithm only needs the size of the sampleset, a counter of the triangles and the number of elements of the stream already stream.

Does the algorithm support edge deletions? At least considering only the algorithms that we implemented (TriestBase and TriestImrpoved) we have no real support for edge deletion. This is is a way a limit of the algorithms. Said that, in the paper that we were given there is also a description of *Triest-FD* which is a further improvement that considers also a fully-dynamical stream in which we have both edge insertion and edge deletion. **Triest-FD** is built upon the concept of Random Pairing (RP) by keeping track of the edges deleted from the sampleset due to deletion in the stream and the overall number of deletions. This information influences the insertion of a new edge in the sampleset and the improved formula for the estimation of the number of triangles.