Data Mining Lab 3: Mining Data Streams

Gibson Chikafa
chikafa@kth.se

Akhil Yerrapragada
akhily@kth.se

1 Assignment Goals

In this assignment we study and implement a streaming graph processing algorithm described in the paper [1] to count triangles in a given graph dataset. The algorithm described in the paper is based upon the reservoir sampling algorithm. We implement both **TRIEST_BASE** and **TRIEST_IMPR** a variant of **TRIEST_BASE** with small modifications that result in higher-quality (i.e., lower variance) estimations

2 Solution overview

We implement the solution in Python programming language.

2.1 TRIEST_BASE

We implement the basic algorithm in the class TRIEST_base. As described in the paper, this class has the following variables:

1. `S`: Edge-sample of size M edges from the stream, where M positive integer parameter. As mentioned in the paper, we assume M $\geq$ 6. `S` is defined as variable **edge_sample**
2. τ global counter for the estimation of the global number of triangles. It is defined as **tau**.
3. `M` the maximum number of edges to keep. Defined in variable **m**
4. `Counters` to compute estimations of the global and local number of triangles
5. Time `t` incremented each time receive an edge

The function **update_counters** increments or decrements global counters for each of the nodes in the subgraph. The function **sample_edge** does the reservoir sampling of determining the random edge that should be deleted from `S` when $t > M$ by flipping the biased coin. **algo_start** starts the algorithm and takes edge-stream. For each edge in the stream we increment **t**, call the **sample_edge**

to determine if we should remove an edge from `S` to add the new edge. At any time $t \geq 0$ the estimation is given by

```
est = max([1,
↪  (self.t*(self.t-1)*(self.t-2))/(self.m*(self.m-1)*(self.m-2))])
```

The number of triangles is then given by **tau * est**.

2.2 TRIEST_IMPR

We implement the improved algorithm in the class TRIEST_improved with the following modifications according to the paper:

1. In the **algo_start**, **update_counters** is called unconditionally for each element on the stream, before the algorithm decides whether or not to insert the edg as hown below:

```
for edge in edge_stream:
    self.t += 1
    self.update_counters(edge)
    if self.sample_edge(edge):
        self.edge_sample.add(edge)
```

2. Never decrement the counters when an edge is removed from **S**. We thus remove the call to **update_counters** in the **sample_edge** function.

3. **update_counters** performs a weighted increase of the counters using

```
weight = max([1,
↪  (self.t-1)*(self.t-2)/self.m/(self.m-1)])
self.tau += weight
self.counters[vertex] = self.counters.get(vertex, 0)
↪  + weight
self.counters[edge.frm] =
↪  self.counters.get(edge.frm, 0) + weight
self.counters[edge.to] = self.counters.get(edge.to,
↪  0) + weight
```

3 Results

3.1 Run times: TRIEST_BASE vs TRIEST_IMPR

Figure 1 and Figure 2, shows run times for different sizes of **M**. As seen **TRIEST_IMPR** performs better than **TRIEST_BASE**.

3.2 Accurancy: TRIEST_BASE vs TRIEST_IMPR

Figures 3 and 4 shows number of triangles for **TRIEST_BASE** and **TRIEST_IMPR** respectively. As seen, **TRIEST_IMPR** the error from the true value is smaller than in **TRIEST_BASE**.

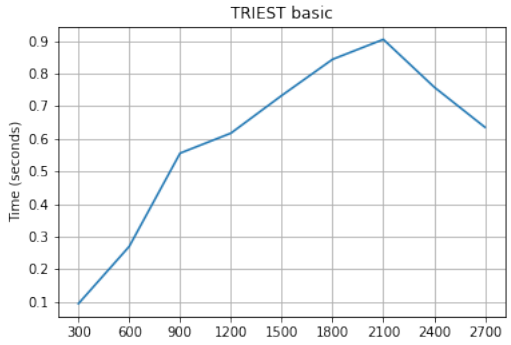

Figure 1: TRIEST_BASE run times

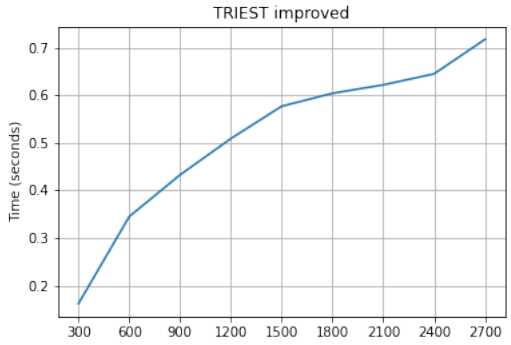

Figure 2: TRIEST_IMPR run times

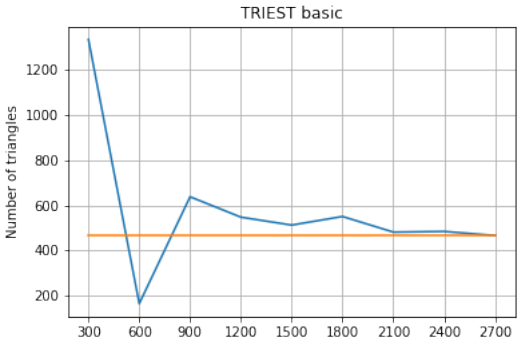

Figure 3: TRIEST_BASE number of triangles

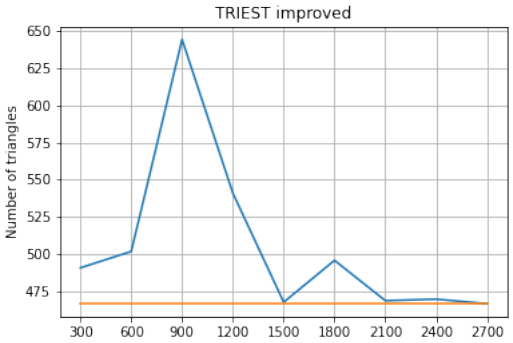

Figure 4: TRIEST_BASE number of triangles

4 Questions

4.1 What were the challenges you have faced when implementing the algorithm?

We found that the paper's explanation of the algorithms was clear and concise, so we didn't face problems with that. However, comparing the results especially the accurancy between the two implementations is difficult since the estimated values varies a lot since flipping the coin is random. Furthermore, running a large dataset, to get a meaningful comparison, on a PC takes alot of time.

4.2 Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.

It is possible to parallize the counting of the triangles using distributed memory. The stream can be partitioned and the number of triangles counted for each partition in each process. The global result is the sum for each partition. For shared memory using threads, some loops can be parallized but since we have alot of shared variables, acquiring locks for the these shared variables can somehow reduce the performance.

4.3 Does the algorithm work for unbounded graph streams? Explain.

The algorithm will work for unbounded streams, since we are using reservoir sampling and random pairing that will allow providing the count at any given time, which means that no matter if the data is unbounded, we will always be able to get an estimation.

4.4 Does the algorithm support edge deletions? If not, what modification would it need? Explain.

The version implemented in this case will allow the deletion with a small modification. When we get the deletion of an edge in the stream, we call the **update_counters** with **operation=remove**. We also need to remove the adge from **S** or **edge_sample**.

References

[1] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. TriÈst: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Trans. Knowl. Discov. Data*, 11(4), June 2017.