Homework 3: Mining Data Streams

Mauro Pungo, Viktoria Sartor

November 2021

1 Introduction

This report briefly presents the results obtained using the algorithm presented on the paper ***"TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size"*** by L. Stefani, A. Epasto , M. Riondato and E. Upfal [1].

The paper discusses an algorithm to count the number of triangles in a graph stream without resorting to the classical methods involving parameters. Furthermore, the algorithm's sole parameter is the amount of available memory, which is fixed [1]. A triangle is a set of edges that interconnects 3 different nodes. They allow to find correlated nodes in a graph.

2 Task 1: Reservoir Sampling

Reservoir sampling is a technique that allows to sample a fixed amount of tuples from the stream. It works by filling the reservoir (a data structure such as an array) and then replace items randomly with a probability whenever a new tuple needs to be inserted and the reservoir is full.

In this context of graph mining, the reservoir is actually a subgraph of the whole streamed graph. When the stream is queried,the data on the reservoir can serve to respond to queries with limited memory. Therefore the bigger the size of the reservoir and the more accurate the queried values are to the real ones.

3 Task 2: Our implementation of the TRIÈST algorithm

The paper presents 3 versions of the algorithm. The first one, *TRIEST-BASE* and the second, *TRIEST-IMPR* only supports edge additions whereas the third version, *TRIEST-FD* also supports edge deletions.

As the reservoir is essentially a graph, we represented it as such. For that, we resorted to a adjacency list representation using dictionaries, in Python. The implementation supports both directed and undirected graphs We represented an edge from node u to v as a tuple (u,v).

4 Extra tasks

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

The correct choice of the data structure for the reservoir was the most troublesome task.
At first, we only thought of the reservoir aa a fixed sized array and so we implemented it as a M-sized deque, which is similar to circular buffer.
Since the algorithms have many insertion and removal operations, our implementation quickly became ineficient for larger values of M.
We then noticed the reservoir is a graph and implemented it as such, the complexity was reduced and the algorithm scaled better for larger values of M.
For example, when using a deque the runtime of the algorithm for the *caida.txt* dataset ($\approx$ 106k edges) went from 6 minutes to around 25 seconds when representing the reservoir as a graph.

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

The algorithm works in a purely sequential manner, reading the stream for one tuple at a time and updating the reservoir and the counters. Therefore we don't think the algorithm can be easily parallelized. Furthermore, even if somehow we managed to parallelize the reading of the stream , synchronism mechanisms would have to be implemented to synchronize read/write operations on the reservoir. Depending on how the synchronization is implemented, there could be no significant speedup observed.

4.3 Does the algorithm work for unbounded graph streams? Explain

Yes it should work for unbounded graph streams since the memory bottleneck is just the size of the reservoir, M. Nevertheless, the count of triangles will only be an approximation of the real count since there is a threshold M_{min} below which the apporximated triangle count is lower than the actual count. Since for unbounded graph streams $\#Edges \rightarrow \infty$ then the error on the triangle count should increase as M decreases, at least for TRIEST-Base.

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

The base algorithm, TRIEST-base, along with its more accurate version, TRIES-IMPR do not support edge deletion. On the other hand, TRIEST-FD does support edge deletion by keeping counters. It works by thinking of an edge deletion as a future addition. The counters keep track of the uncompensated deletions for edges on the reservoir when the edge deletion occured on the stream.

5 Results

In order to test the performance of the algorithm we used graph datasets from `https://snap.stanford.edu/data/#web` . The results presented are from the dataset CAIDA although we also extensively tested the algorithm with the smaller 14.4k edges GrQc dataset and the 2.3M edges Stanford Web Graph dataset.

5.1 TRIESTBase

Results see below.

5.2 TRIEST-Impr

Results see below.

6 Build instructions

Syntax: python homework3.py [ARG_FLAG1] [ARG_VALUE1]
For further help just run *python homework3.py -h* Run script *homework3.py* in the same directory where the data-directory is stored.

Argument parser:

```
--dataset choices: 'grqc.txt', 'caida.txt', 'stanford.txt'

--variant choices: 'base', 'improved'

--memory default: 15000

All flags with double hyphens!
```

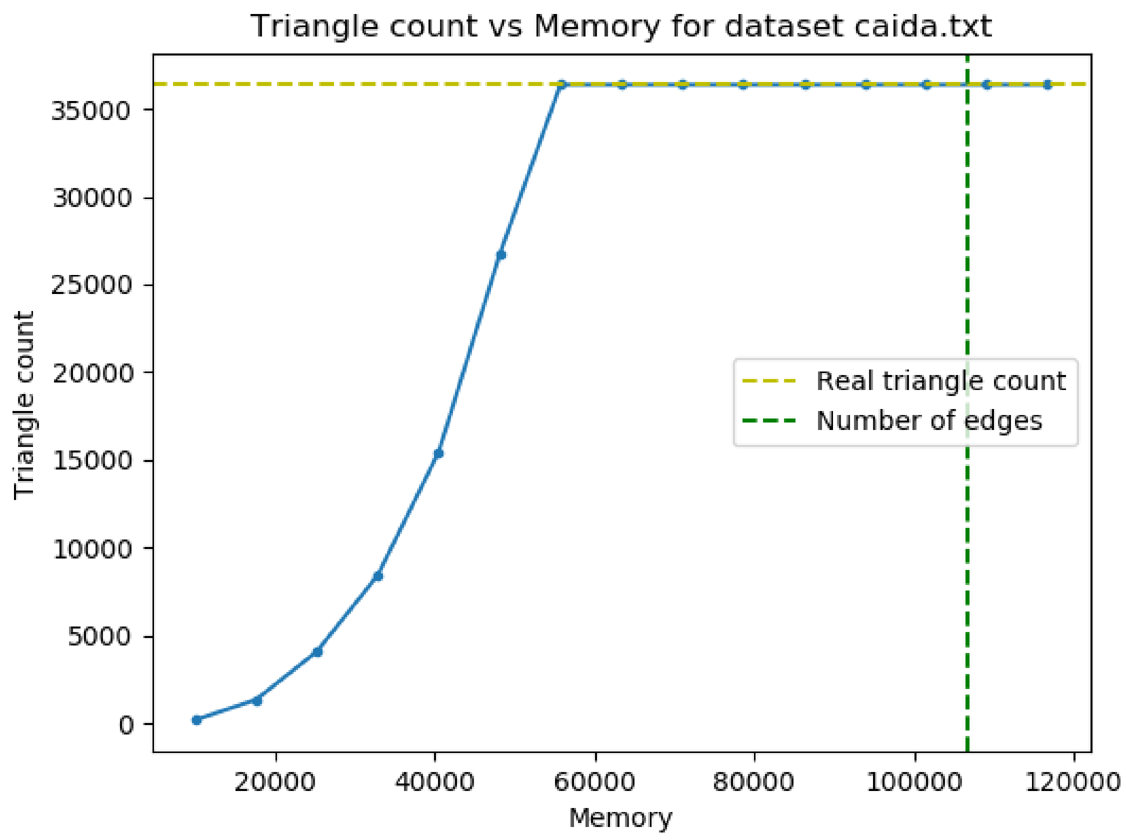

Figure 1: TRIESTBase Triangles vs. Memory

References

[1] Lorenzo De Stefani et al. “TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size”. In: *ACM Trans. Knowl. Discov. Data* 11.4 (2017), 43:1–43:50. DOI: `10.1145/3059194`. URL: `https://doi.org/10.1145/3059194`.

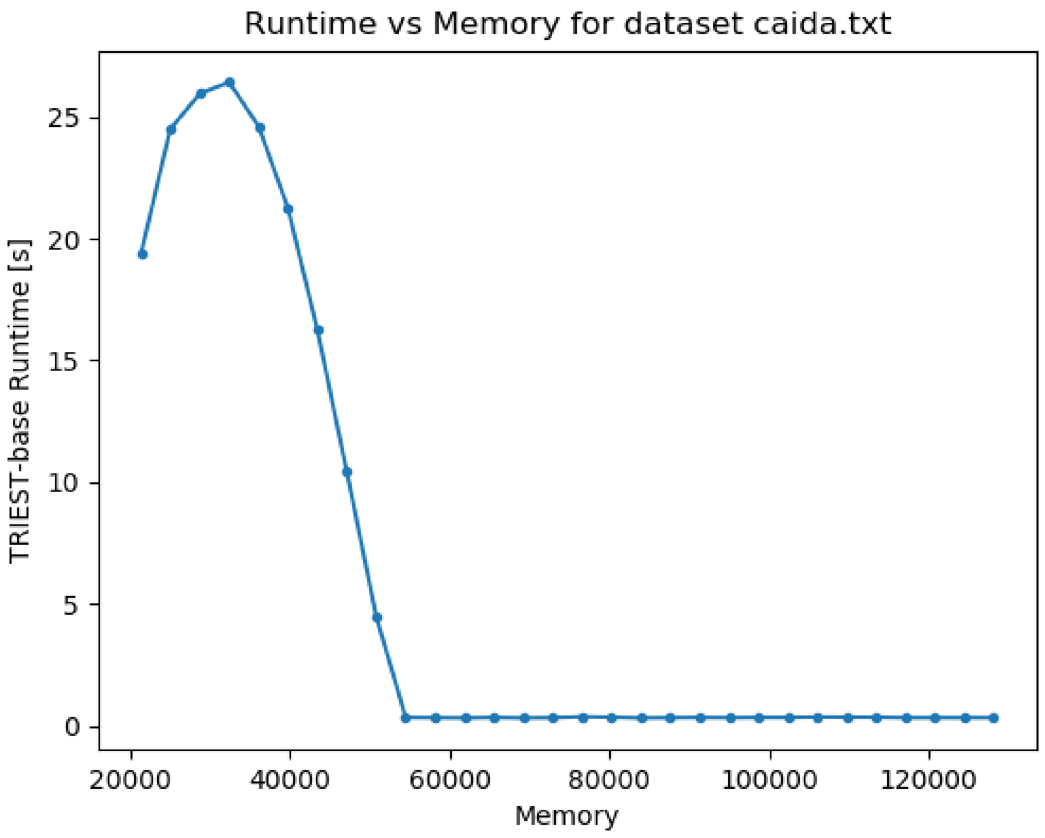

Figure 2: TRIESTBase Runtime

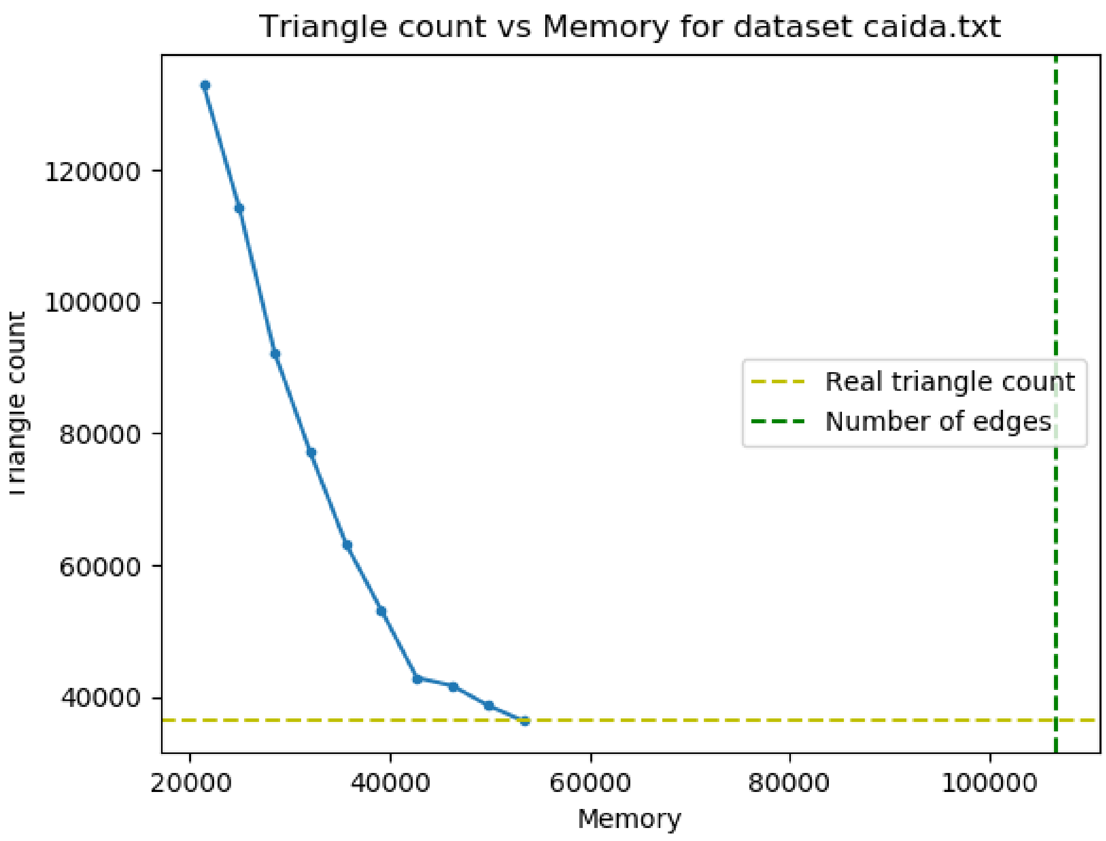

Figure 3: TRIEST-Impr Triangles vs. Memory

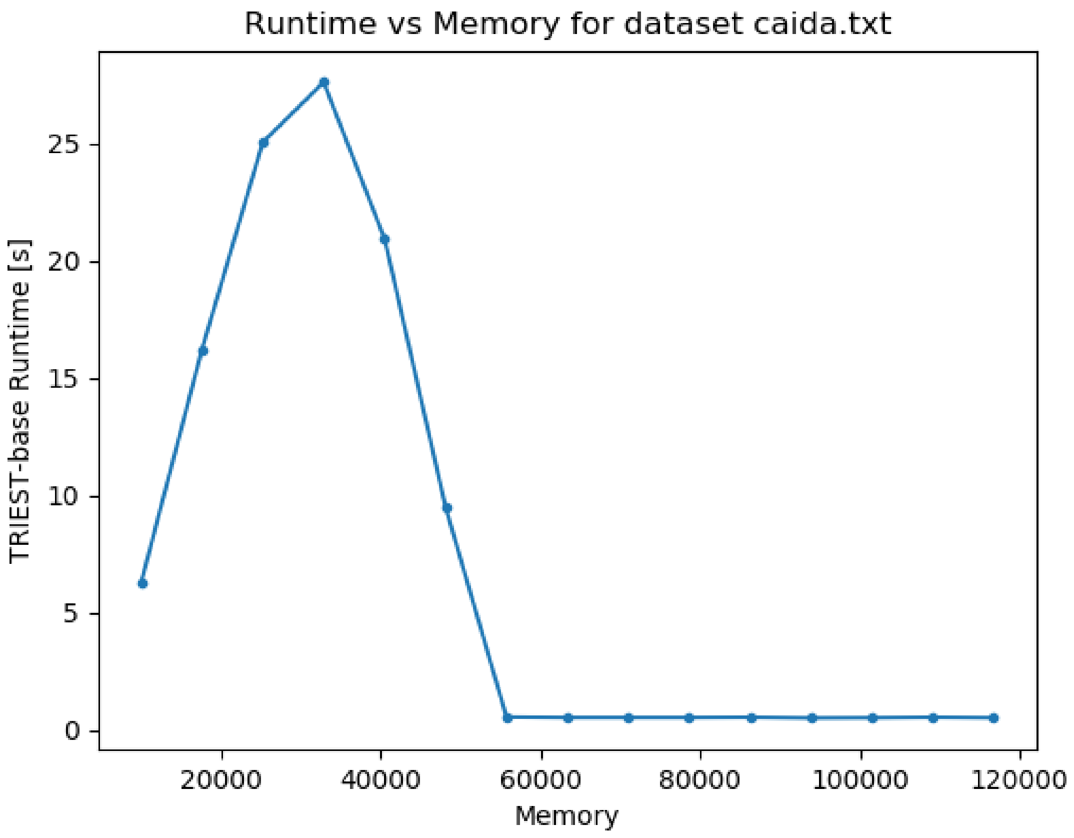

Figure 4: TRIEST-Impr Runtime