

Data Mining - Homework 3

Robert-Andrei Damian and Alice De Schutter
Mining Data Streams

25 november 2021

Task

Study and implement a streaming graph processing algorithm described in one of the above papers of your choice. In order to accomplish your task, you are to perform the following two steps:

- First, implement the reservoir sampling or the Flajolet-Martin algorithm used in the graph algorithm presented in the paper you have selected;
- Second, implement the streaming graph algorithm presented in the paper that make use of the algorithm implemented in the first step.

To ensure that your implementation is correct, you are to test your implementation with some of the publicly available graph datasets (find a link below), and present your test results in a report.

1 Detailed Information

Apache Kafka was used to simulate a data stream with different threads (default value is 4 threads). We implemented reservoir sampling and the stream graph algorithm presented in the paper called **TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size**. We used the dataset that can be found via this link: High-energy physics citation network.

The algorithm presented in the paper is summarized in Figure 1.

Algorithm 1 TRIÈST-BASE

```
Input: Insertion-only edge stream  $\Sigma$ , integer  $M \geq 6$ 
1:  $\mathcal{S} \leftarrow \emptyset$ ,  $t \leftarrow 0$ ,  $\tau \leftarrow 0$ 
2: for each element  $(+, (u, v))$  from  $\Sigma$  do
3:    $t \leftarrow t + 1$ 
4:   if SAMPLEEDGE( $(u, v), t$ ) then
5:      $\mathcal{S} \leftarrow \mathcal{S} \cup \{(u, v)\}$ 
6:     UPDATECOUNTERS( $+, (u, v)$ )

7: function SAMPLEEDGE( $(u, v), t$ )
8:   if  $t \leq M$  then
9:     return True
10:  else if FLIPBIASEDCOIN( $\frac{M}{t}$ ) = heads then
11:     $(u', v') \leftarrow$  random edge from  $\mathcal{S}$ 
12:     $\mathcal{S} \leftarrow \mathcal{S} \setminus \{(u', v')\}$ 
13:    UPDATECOUNTERS( $- , (u', v')$ )
14:    return True
15:  return False

16: function UPDATECOUNTERS( $(\bullet, (u, v))$ )
17:   $\mathcal{N}_{u,v}^S \leftarrow \mathcal{N}_u^S \cap \mathcal{N}_v^S$ 
18:  for all  $c \in \mathcal{N}_{u,v}^S$  do
19:     $\tau \leftarrow \tau \bullet 1$ 
20:     $\tau_c \leftarrow \tau_c \bullet 1$ 
21:     $\tau_u \leftarrow \tau_u \bullet 1$ 
22:     $\tau_v \leftarrow \tau_v \bullet 1$ 
```

Figure 1: Summary of TRIEST algorithm

Instructions on how to build and run the program

- `systemctl start kafka` (which starts the Kafka service)
- `python3 stream-processor.py` (which starts listening for messages on Kafka)

In a separate terminal:

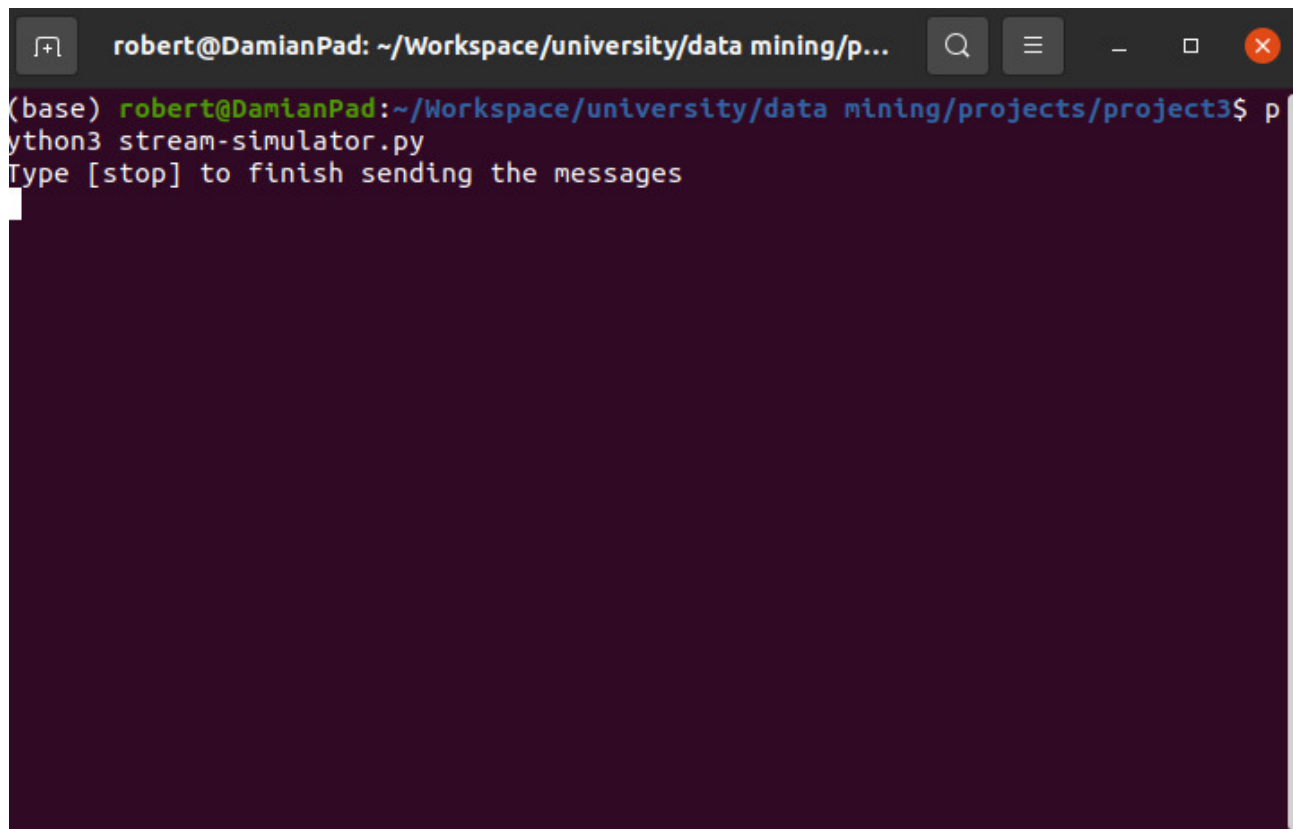
- `python3 stream-simulator.py`

2 Results

The results shown below are found with default values:

- $CONST_MAX_EDGE_COUNT = 10000$ (default value)
- $CONST_THREAD_COUNT = 4$ (default value)

Task 1:

A terminal window titled "robert@DamianPad: ~/Workspace/university/data mining/p..." with search, menu, and window control icons. The prompt is "(base) robert@DamianPad:~/Workspace/university/data mining/projects/project3\$". The user has entered "python3 stream-simulator.py". The output is "Type [stop] to finish sending the messages". The terminal background is dark purple.

```
(base) robert@DamianPad:~/Workspace/university/data mining/projects/project3$ python3 stream-simulator.py
Type [stop] to finish sending the messages
```

Figur 2: stream-simulator interface

```
(base) robert@DamianPad:~/Workspace/university/data mining/projects/project3$ python3 stream-processor.py
New global triangle count: 1
New global triangle count: 2
New global triangle count: 3
New global triangle count: 4
New global triangle count: 5
New global triangle count: 6
New global triangle count: 7
New global triangle count: 8
New global triangle count: 9
New global triangle count: 10
New global triangle count: 11
New global triangle count: 12
New global triangle count: 13
New global triangle count: 14
New global triangle count: 15
New global triangle count: 16
New global triangle count: 17
New global triangle count: 18
New global triangle count: 19
New global triangle count: 20
New global triangle count: 21
New global triangle count: 22
New global triangle count: 23
New global triangle count: 24
```

Figure 3: Triangle count terminal output

```
robert@DamianPad: ~/Workspace/university/data mining/...
New global triangle count: 1576
New global triangle count: 1575
New global triangle count: 1577
New global triangle count: 1575
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1578
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1574
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1577
New global triangle count: 1576
New global triangle count: 1577
New global triangle count: 1578
New global triangle count: 1579
```

Figure 4: Triangle count terminal output (converges)

Task 2 (bonus):

1. **What were the challenges you have faced when implementing the algorithm?**

We found it difficult to simulate a data stream. We ended up using Apache Kafka but it took us a while because we had never used it before.

Additionally, we first wanted to use Apache Spark to implement the reservoir sampling algorithm efficiently. However, Spark offers a high level API and this didn't allow us to tamper with the messages observed on the Kafka socket. Spark does have an inbuilt reservoir sampling method but we wanted to do our own implementation. Therefore, we decided to implement the algorithm with vanilla Python.

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

We do not believe that the algorithm can easily be parallelized. This is due to the high frequency with which the program accesses two main objects: the edge set (edgeSet) and the triangle counter (counters). Thus if we were to try parallelizing the program, all the threads would have to wait for each other to perform the read /write operations on these objects. As the algorithm consists mostly of these read /write operations, we do not expect much to be gained from the parallelization. Moreover, according to Amdahl's law, this may even prove detrimental due to the overhead introduced by managing the separate threads.

3. **Does the algorithm work for unbounded graph streams? Explain.**

Yes, the algorithm works for unbounded graph streams because it leverages the benefits of reservoir sampling: allowing us to store only a small sample (CONST_MAX_EDGE_COUNT) of the whole graph at any given time. Although we do not store the whole graph S , we have reason to believe that our sample is a good representation of the graph in its entirety in the scope of answering the research question.

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

Yes. Edge deletions is implemented in our algorithm. The deletion happens when we drop an edge according to reservoir sampling. In addition, we can modify the messages in the data stream to include an operation (addition or deletion).