Introduction

The code used in this lab reproduces the streaming algorithms TRIÈST-BASE and TRIÈST-IMPR found in "L. De Stefani, A. Epasto, M. Riondato, and E. Upfal, TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size, KDD'16". To test and evaluate the implementation we used the dataset Social circles: Facebook.

How to run

- Download the dataset from this (link)
- Ensure that Python3 and SciPy is installed
- Go into the Mining Data Streams map
- Run instruction

```
python3 triest.py
```

Execution and Results

The dataset contained 4039 nodes, 88234 edges and 1612010 triangles. Since the TRIÈST algorithm is used to approximate the number of triangles, we will try to compare the number of triangles found by the Triest Base and Improved algorithm to 1612010.

In our implementation we used **M=5000** for TRIÈST-BASE and **M=1000** for TRIÈST-IMPR and got the results:

```
Running the algorithm with M = 5000.
The estimated number of triangles is 1622055.6896721134.
Time to run Triest Base: 11.37 seconds
Running the algorithm with M = 1000.
The estimated number of triangles is 1620192.6704844844.
Time to run Triest Improved: 6.01 seconds
```

As we can see, the results for both are very similar to the actual number of triangles. However, one has to keep in mind that part of the algorithm is random, hence the results will not always be the same. The choice of **M=5000** for TRIÈST-BASE and **M=1000** for TRIÈST-IMPR was based on trial and error when testing different values for M and based on what gave desirable results.

Questions

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

The algorithm was well documented and the pseudo code was easy to understand but some parts were still kind of difficult to implement. And since we both are not that familiar with Graph implementations, it took a bit of time to learn how it works.

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

Not really since the triangle count depends on the current state of the sample set which is updated sequentially. Since the data comes in a stream which follows a sequential order it limits the opportunity for dividing the workload across multiple processors.

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

Yes. This algorithm in principle is designed to handle dynamic graphs where the number of events grows continuously over time. We maintain a fixed size of sample item/memory with reservoir sampling, so the required resources stay the same as the number of edges increases. One limitation is edge deletion which will be addressed in the next question. While in principle the algorithm can work indefinitely, as with any sampling algorithm, the result can be biased which might require other techniques to mitigate.

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

No. The algorithm used in this work is from the TRIÈST suite, TRIÈST-BASE, and the improved version TRIÈST-IMPR which is technically designed for insertion stream only. The last algorithm designed for fully dynamic graphs (TRIÈST-FD) can handle edge deletions. The TRIÈST-FD introduces two new variables to track **uncompensated deletions** for edges that were in S ($\mathbf{d_i}$) & not in S ($\mathbf{d_o}$). The idea behind this is one edge deletion will be compensated by another insertion in the future.

- Adding $\mathbf{d_i}$ and $\mathbf{d_o}$ on initialization
- If a new deletion edge is in S, increase $\mathbf{d_i}$, if not increase $\mathbf{d_o}$.
- If $\mathbf{d_i} + \mathbf{d_o} = 0$, it means no uncompensated edge -> perform normal reservoi sampling, else adds edges with a probability proportional to $\mathbf{d_i} / (\mathbf{d_i} + \mathbf{d_o})$, reflecting the need to compensate for previous deletions.
- The number of $\mathbf{d_i}$ and $\mathbf{d_o}$ are deducted accordingly with insertion.