DATA MINING - HOMEWORK 3
Mining Data Streams

SERGHEI SOCOLOVSCHI, ANGEL IGARETA

23 November 2020

Main Tasks

The goal of this project is to study and implement TRIÈST [1]*, a* streaming graph processing algorithm which aims to count the local and global triangles in fully dynamic streams. For the purpose of the assignment, we would focus on implementing both the basic and improved version of the algorithm.

To accomplish the task, the following two steps will be performed:

1. First, implement the reservoir sampling or the Flajolet-Martin algorithm used in the graph algorithm presented in the selected paper.
2. Second, implement the streaming graph algorithm presented in the paper that makes use of the algorithm implemented in the first step.

Tools & Dataset

This project has been implemented using *Python* and *PySpark*, to be able to have a parallelized version of the algorithm when running in a distributed cluster. Additionally, the dependencies *NumPy*, *random*, and *urllib.reques*t (to download the dataset from remote storage), were used for the implementation.

The selected graph dataset was retrieved from Stanford Network Analysis Platform (SNAP) and it was previously used in other publications [2]. The dataset contains a collaboration network of papers published on arXiv in the AstroPhysics category.

Implementation

The implemented methods and classes are the following:

- **DictAccumulator and DictAccumulatorMethod**: Classes used to create PySpark accumulators for the sample dictionary and local triangle counters, to make the implemented algorithm parallelizable in a distributed cluster.

- **get_current_neighborhood**: Given an edge, return the neighborhood of both vertex1 and vertex2 among the sample dictionary.

- **add_edge_to_samples**: Add a new edge to the sample dictionary by adding to the array with the key ‘vertex1’ the ‘vertex2’ and vice-versa.

- **remove_edge_from_samples**: Remove an edge from the sample dictionary, if a vertex that formed the edge does not have any other connection, also remove the key from the dictionary.

- **flip_biased_coin**: Return true with probability **p**. Used for imitating the biased coin flip with probability (sample_dimension / elements_seen).

- **update_local_triangle**: Updates local triangle counts of vertices of the edges that have been added to or removed from the sample collection.

- **get_commons neighbors**: Look for the sample neighborhood of each of the vertices that formed the received edge and return the common neighborhood.

- **update_counters**: Calculate the common neighbors of an edge and for each neighbor, update the global triangle counter, the local counters both of the vertices and the neighbor. In trièst-impr, it also performs a weighted increase of the counters by using. $\eta(t) = max\left\{1, \frac{(t-1)(t-2)}{M(M-1)}\right\}$as weight, where *t* is *elements_seen* and *M* is *sample_dimension*.

- **sample_edge**: Implements the reservoir sampling technique that accepts a new sample if there is still space available, otherwise, with the probability of *sample_dimension* / *elements_seen,* it might replace a random sample with the new one.

- **triest_basic**: Wrap the main logic of the Trièst algorithm, using all the helper functions described above.

- **triest_impr**: Wrap the main logic of the Trièst algorithm, adding small modifications that result in higher-quality (i.e., lower variance) estimations.

The global variables:

- **elements_seen**: An accumulator that tracks how many edges have been received from the stream.
- **sample_dimension**: Defines the dimension of the sample container.
- **total_triangles**: An accumulator that keeps track of the total number of triangles observed in the graph.
- **edge_samples**: A dictionary where the sampled edges are stored. The keys are the vertices and the values are the lists of neighbors.
- **triangles_per_edge**: A dictionary that tracks the number of triangles each edge is part of.

Execution

The code is in the format of a Jupyter Notebook, which can be executed in Google Colab or locally if Jupyter Notebook is installed. Before starting running the algorithm, the size of the reservoir memory can be defined by initializing the *sample_dimension* variable to the desired value.

To run the trièst basic, the following command can be used:

```
triest_basic(graph_edges_list)
```

The resulting triangles would be stored in the global variable total_triangles, along with the local triangles per edge and the final edge samples.

To run the trièst improved implementation, the following method can be called:

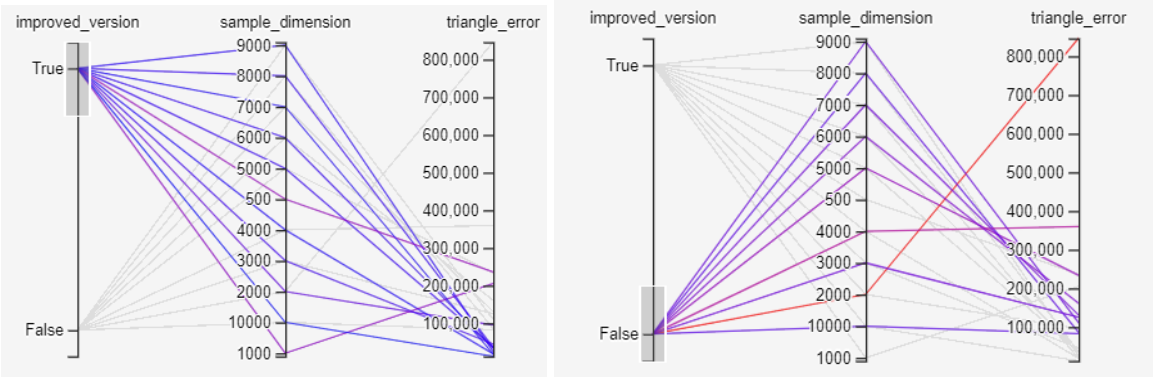
```
triest_impr(graph_edges_list)
```

Results

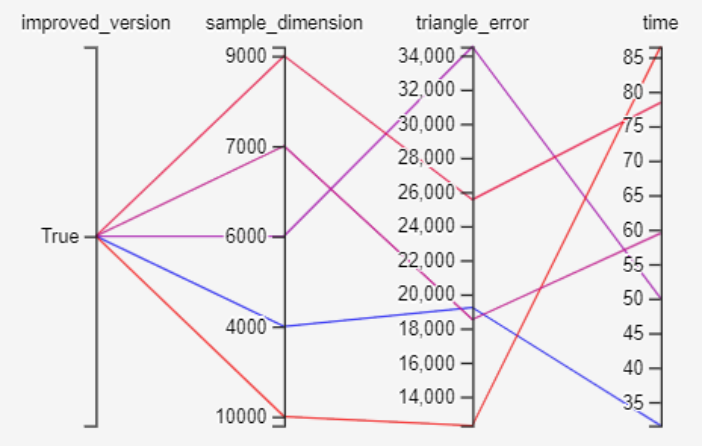
In order to find the most suitable dimension for the sample list, we decided to implement a basic hyper tuning of that parameter together with a parameter that indicates if using the improved version of the Trièst algorithm. The list of sample dimensions selected was: *[100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 7500, 10000, 15000].* To achieve accurate results, we run each combination of parameters *3* times, which is also parametrized. Finally, to compare the results we calculated a basic measure for each combination, called 'triangle_error', which consists of the absolute difference between the expected triangles of the dataset and the estimated triangles. Also, we save the algorithm execution time for each combination.

The results for a total number of 84 executions are:

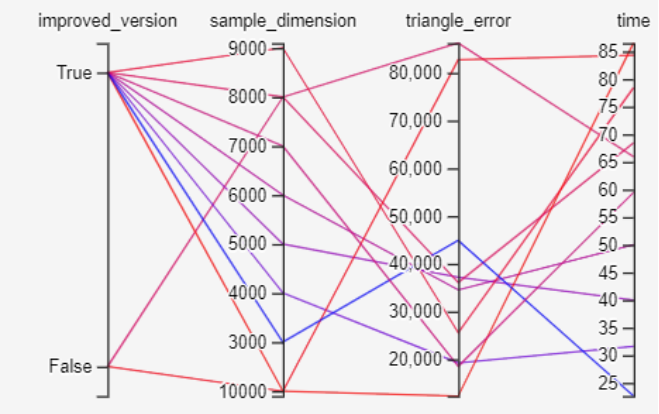
- By comparing the triangle_error on the combinations using the improved versions with the one not using it, the results are significantly better for the improved implementation.



- The top 5 results were obtained by using the sample sizes 10000, 7000, 4000, 9000, and 6000, ordered from the top-1. These results were 12303, 18520, 19223, 25544 and 34509, again ordered from the top-1.



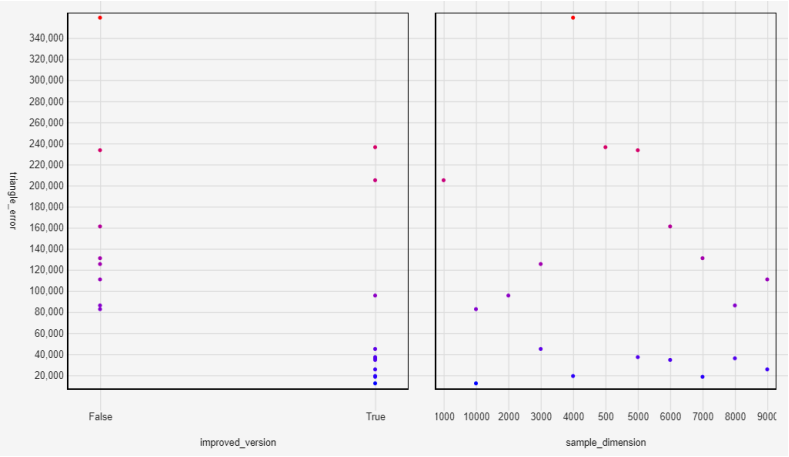
- Finally, regarding the time, the best top 10 results were between 22 and 86 seconds, on a dataset with a total of 198110 edges.

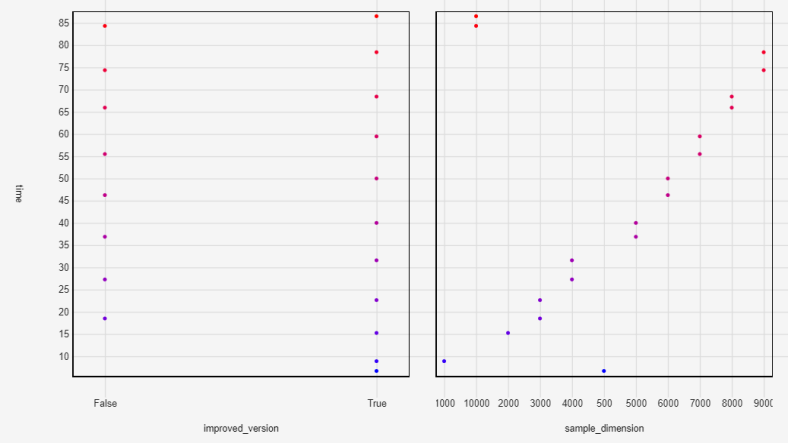


- According to the ratio triangle-error time, the best results was, with one of the three runs reaching a triangle_error of 4048:

improved_version	sample_dimension	time	triangle_error
True	4000	31.572	19223

Hence, in conclusion, the improved version is significantly better than the basic one and slightly slower. Also, the results indicated that when having a bigger sample size the results are usually better.





Additional Questions

Q: What were the challenges you have faced when implementing the algorithm?

The basic and improved versions of the Trièst algorithm were clearly explained in the paper and did not pose any problems with their implementation. However, the chosen dataset required some preprocessing before exploitation, such as the removal of duplicates and autoloops. Additionally, there was an attempt of implementing a parallelized version of the algorithm using PySpark, which added difficulty in terms of implementation, even though all the tests were performed using only one cluster.

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

Yes, however, it requires careful management of the shared variables, such as the time or elements seen and both local and global triangle counters, in order to track the stream progress. These variables could be updated in parallel, by using special techniques that provide concurrent modifications such as the accumulators we used in our implementation with Spark.

Q: Does the algorithm work for unbounded graph streams? Explain.

Yes, since it is possible to query an estimation of the global or local triangle counts at the time t. For the basic version of the algorithm, the returned result will be $\varepsilon \cdot \tau$, where $\tau$is the triangle counter at time t and $\varepsilon = \frac{t(t-1)(t-2)}{M(M-1)(M-2)}$. While the improved version will return directly the τwithout any estimation error coefficient.

Q: Does the algorithm support edge deletions? If not, what modification would it need?

In the case of Trièst algorithms, only the fully-dynamic version of the algorithm supports external edge deletions. It would require the usage of random pairing, which is an extension of reservoir sampling to handle fully-dynamic streams with insertions and deletions.

References

[1] 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.
[2] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007.