Data Mining

Mining Data Streams

Juan Álvarez Fernández del Vallado & Zhaopeng Tao

23 November 2020

1. Objective of this project

This project consists of counting triangles (from now on closed *wedges*) in a very long graph. In this case, we consider a graph so big that we implement streaming processing to process it and count the wedges.

The project is completely inspired by this paper, where the author counts wedges and the transitivity using the *birthday paradox*.

2. Running the project

To run this project we need the Jupiter notebook and the data to be in the same folder and specify this route inside the Jupiter notebook code. Moreover, the notebook uses Python3 as runtime engine and the following packages:

- *networkx:* Used for implementing the subgraph from the main graph which is going to have updated edges all the time and which will have some subtracted edges as well.
- *random:* Used for finding random numbers following a normal distribution between 0 and 1. This is required for implementing the reservoir sampling.
- *numpy:* Used for creating empty arrays with a given size. Needed for storing the edge reservoir and the wedge reservoir.
- *re:* Used for parsing an edge from the main dataset using a RegEx expression.
- *datetime:* Used for setting a limit for computing the graph. We use this because it takes a lot of time to compute the whole graph.

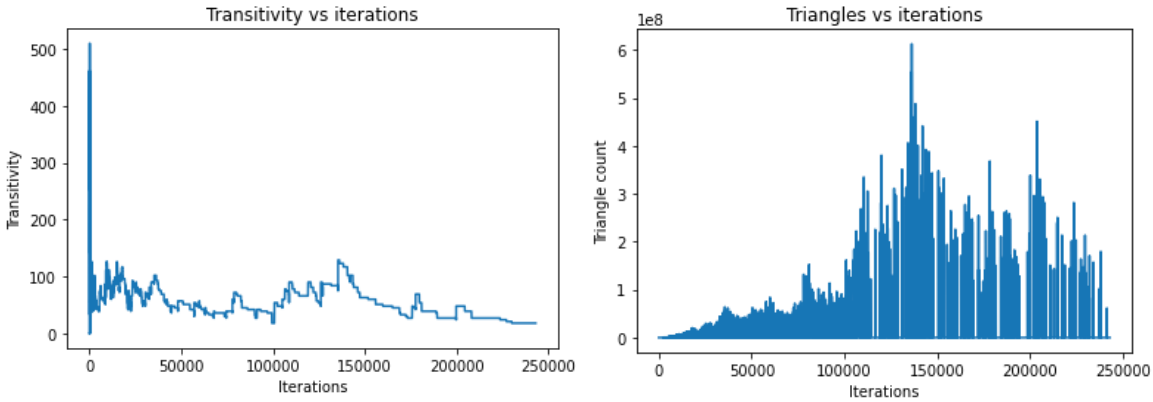
This project uses many loops which greatly affects the running speed of the algorithm. Therefore, bear in mind it takes time to compute all the wedges from the dataset. Also, it is not recommended to use a smaller graph

dataset since the algorithm uses two very large buffers, edge reservoir and wedge reservoir.

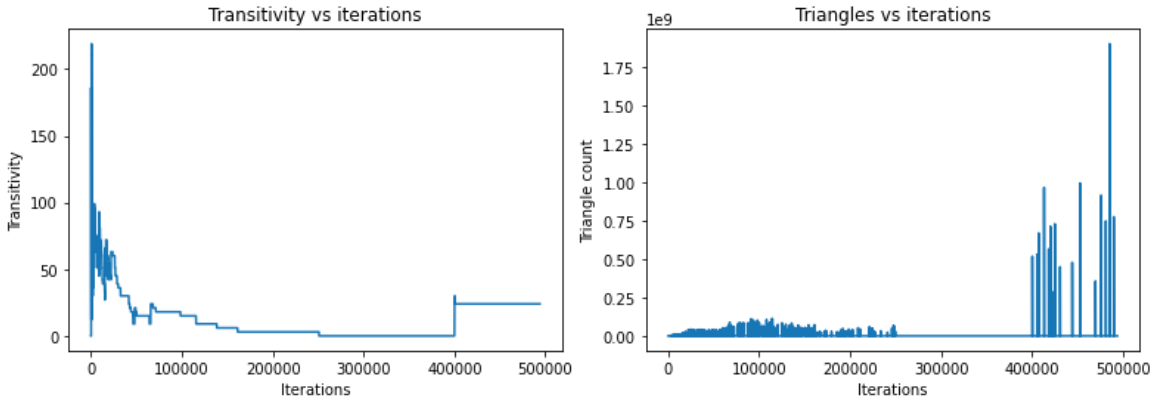
However, we have modified the size of those two large buffers to study how do these parameters affect the overall count. The results for these variations can be found in the next section.

Inside the submitted folder there will be a Python notebook and two datasets. We only use one of them, the large one.

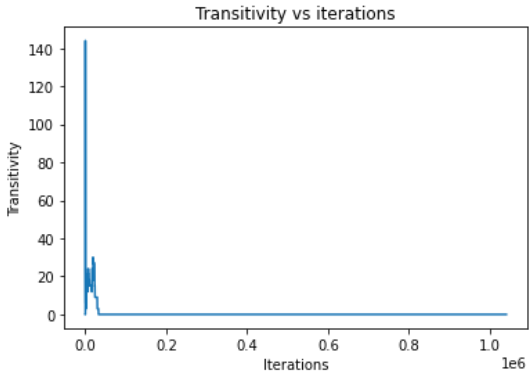
3. Results

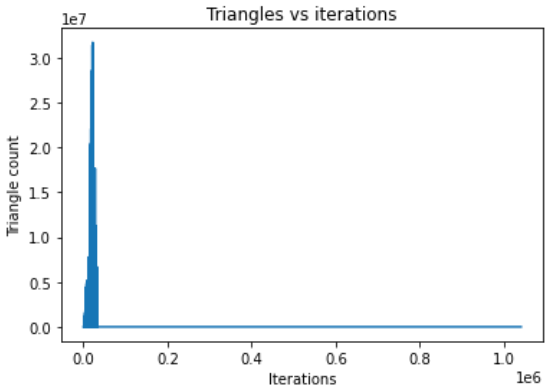


The plots above represent the number of iterations (processed edges) against the transitivity and triangle count. The buffer sizes chosen where 200.



The plots above represent the number of iterations (processed edges) against the transitivity and triangle count. The buffer sizes chosen where 100.





The plots above represent the number of iterations (processed edges) against the transitivity and triangle count. The buffer sizes chosen where 50.

We can see from the plots how crucial the choice of the buffers can be.

Initially the buffer sizes were 20000 because in the paper we implemented it was said the best results were obtained using these values of the sizes. However, the greater the value of the buffer, the more iterations the algorithms does. So, we decided to reduce these values. Actually, we have let the algorithm run for over 6 hours with buffer sizes 20000 and we did not get any result because the algorithm was running.

4. Extra bonus points question

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

Implementing the algorithm required, firstly, reading the paper many times. There were some very-specific design characteristics related to implementation that were not properly explained and made it harder to understand the algorithm. In whole, this made it very hard to get an overall intuition of how the algorithm worked.

In terms of implementation, it was hard to get a result given the amount of loops the algorithm needed to compute the wedges and on the other hand one can not just simply reduce the size of the graph because the buffers are quite big and they need to be full to be fully utilised by the algorithm.

There was another issue with this project which had to do with the programming language. In the beginning, we decided to implement this algorithm in Spark, taking advantage of the Streaming framework offered in Spark. However, as many difficulties arose we decided to stay in Python and implement the streaming nature of the algorithm using the *open* function built-in.

Finally, another difficulty was getting the wedges from the subgraph because you needed to go through all the nodes and find those closed wedges connected to the current streamed edge.

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

As mentioned before we tried to implement this project using Spark Streaming functionality but many issues arose. So, probably it is possible but it is not easy.

Moreover, if you had the dataset in a distributed database and you wanted to implement a read line-by-line architecture (just like we did), you would have to collect every single time a new edge and this takes some time because you have to access the information from the datanodes containing that edge. Furthermore, it would be difficult to keep track of which edges had been processed because having the data distributed does not guarantee ordered items.

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

Yes it does. Actually, the dataset we are considering is so big that we could almost consider it unbounded. The issue here is that the count of closed wedges can change a lot.

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

Yes, in fact we delete edges constantly to keep the same edges in the subgraph as in the fixed edge reservoir.