Report 3: Mining Data Streams

Laila Niazy, Svenja Raether

November 26, 2019

1 Introduction

The objective of assignment 3 is to implement one of the streaming graph processing methods and corresponding algorithms that make use of the stream mining algorithms presented in the course. For our solution we decided to implement the reservoir sampling technique together with the the graph algorithm presented in the TRIEST paper. This implementation counts the local and global number of triangles in a streamed graph given a fixed memory size. It can be implemented in three versions: basic, improved and fully dynamic. For this assignment we decided to implement all three versions. The report describes the function and structure of each implementation and compares the results in the evaluation section.

2 Description of our solution

In this section, we will describe our implementation, which is divided into three main parts: TRIEST Base, TRIEST Improved and TRIEST Fully Dynamic.

However, before we go through each function, we will give an overview of our git repository structure. We have two folder in the first level and one python-file. The folder "Dataset" contains the three datasets, which we decided to test our algorithms on and the folder "Report" contains the report. The file, which is called *main.py*, contains the class Triest, which contains all three different version of the algorithm. Images contains the plots used for evaluation and *plotting.py* contains the code for plotting.

Once a triest object is created from the Triest class, the default parameters are initialized. The user selects the memory size M (number of edges that can be kept in memory) and the path to the data set. Furthermore, S (the subset of the graph), global tau (global number of triangles), local taus (local number of triangles and neighbourhood counters for each node) and di, d0, w, s, k, and p are initialized by default.

2.1 TRIEST Base

Method:

triest_base To start the base algorithm, one must call the triest_base function. This iterates over the lines of the data set and ignores comments. For each line it extracts the nodes u and v from the given edge. Then, the counter t gets incremented by 1 and *sample_Edge* is called to check whether the memory is already filled. If True is returned, the new edge is added and the *update_Counter* method is called with a '+' operator.

Once all iterations are over, epsilon is calculated. If the time counter t exceeds the memory size M, we multiply epsilon by tau and return this estimation. Otherwise, the estimation will be tau.

sample_Edge This method tests whether t is smaller then M. If the condition is fulfilled, True is returned which leads to adding the edge to the sample set S. Otherwise, we flip a coin using the coin_flip method with the probability of $\frac{M}{t}$. If the coin_toss is smaller then $\frac{M}{t}$ we remove a random edge from S, call *update_Counter* with the '-' operator and return True in order to add the new edge to S. Else, we return False and keep the existing S meaning that the new edge will not be added.

coin_flip The coin flip method takes $\frac{M}{t}$ as the probability and compares it with a random float value between 0 and 1. If the random value is smaller then the given probability, True will be returned, otherwise False.

update_Counter This method uses *get_neighbours* to get the neighbours of both nodes of the current edge in S. It calculates the intersections between the two neighbourhoods in order to find out how many triangles exist in S for this edge. The number of intersections will be added to global and local tau for u and v. Local tau c will be incremented by 1 for each c (common neighbour).

get_neighbours This function iterates over each edge of S and tests whether the edge is connected with either u or v. If so, the neighbour will be added to the neighbourhood set of the corresponding node u or v. Both neighbourhood sets are returned by the function.

2.2 TRIEST Improved

In order to implement the improved version the three modifications described in the TRIEST paper were applied to the base implementation.

1. In TRIEST improved, the *update_Counter* is moved before the if condition *sample_Element* is checked. This way the counter will be called

unconditionally for each element on the stream.

2. The *sample_Edge_improved* will not call *update_Counter* with a negative operator '-' which means no edge will be removed from S.
3. *update_Counters_improved* uses a weighted increment of n_t instead of 1.

2.3 Fully Dynamic TRIEST

In order to implement the fully dynamic version described in the TRIEST paper, the reservoir sampling was extend to handle edge deletions as well and is now called random pairing. The modifications, we applied to the base implementation to construct the fully dynamic triest will be described now.

triest_dynamic To start the dynamic algorithm, one must call the triest_dynamic function. This iterates over the lines of the data set and ignores comments. For each line it extracts the nodes u and v from the given edge. Then, the counter t gets incremented by 1. Further, we use random float value between 0 and 1, if the random value is smaller then 0.9, the edge will be assigned '+' operation, otherwise it will be assigned '-' as an operation. Depending on the operation, we fulfill a different if-condition. If the operation is '+' and *sample_Edge* is true, then *update_Counter* is also called to update the local and global counters. If the extracted edge is in the set of S, *update_Counter* is called to update the local and global counters and the variable d_i is incremented by 1, which is a counter that keeps track of the number of uncompensated edge deletions involving an edge that was in S at the time the deletion for the edge was on the stream. Otherwise only the variable d_o is incremented by 1, which is a counter that keeps track of the number of uncompensated edge deletions involving an edge that was not in S at the time the deletion of the edge was on the stream.

Once all iterations are over, we calculate p for the last t. p is 0 if M is smaller than 3. Otherwise p follows the calculations in the paper and gets returned as the estimation.

sampleEdge_Fd This method tests whether d_o plus d_i is equal to 0. If the condition is fulfilled, we go into another if condition set, which checks in the first if whether the length of the set of S is smaller than M, if it is true, the edge is added to the set S and True is return as output of this method. The next if-else inside the big if-loop, checks if *coin_flip* method with input $\frac{M}{t}$ returns a true and then selects an edge randomly from S, remove it, add the new edge, call *update_Counter* with the operation '-' to update the counters and returns True as output of this method. Now going to the big if condition we had, if the first condition is not fulfilled, we test the next if

condition using the method *coin_flip* with the input $\frac{d_i}{d_i+d_o}$. If its true, the edge is added to the set, d_i is decremented by 1 and True is returned as the output of this method. Finally if the condition using the method *coin_flip* is not fulfilled, we decrement d_o by 1 and return False as the output of this method.

3 Evaluation

3.1 Data sets

We decided to test the implemented algorithms on three different data sets. Those mainly differ in size and volume.

Data set	Format	Triangle Count	Size (vertices)	Volume (edges)
CAIDA	Undirected	36,365	2,628	26,475
Hamsterster friendships	Undirected	16,750	1,858	12,534
Contiguous USA	Undirected	57	49	107

Table 1: Properties of the selected data sets

3.2 Tests

To test all various versions of TRIEST, we used all three datasets and varied the variable M, which determines how much of the stream will be stored before we start deleted edges in the saved set to free up memory space. In the three figures below, the x-axis depicted the value of M while the y-axis represents the number of estimated triangles in the set. The legend shows which curve belongs to which algorithm. In Fig.1, we can see the results of the Hamsterster friendships dataset. The actual number of triangles was estimated correctly the more memory we use, which corresponds to higher M. This behavior can be seen cleary, when looking at the curve for base and improved TRIEST. In addition, as we can seen the dynamic version of TRIEST has the lowest count on global triangles and this is expected, because the algorithms handles edge deletions as well, which are produced with a probability of 10%. Additionally, the estimation for the dynamic implementation does not go along with the expected estimation once the M is bigger than s. The same behavior can be observed in both Fig.2 and Fig.3. We also ran all three algorithms multiple times and plotted the average of four estimations for the various M. The results can be seen in Fig.4 for the Hamsterster friendships dataset, in Fig.5 for the Contiguous USA and finally in Fig.6 for the CAIDA dataset.

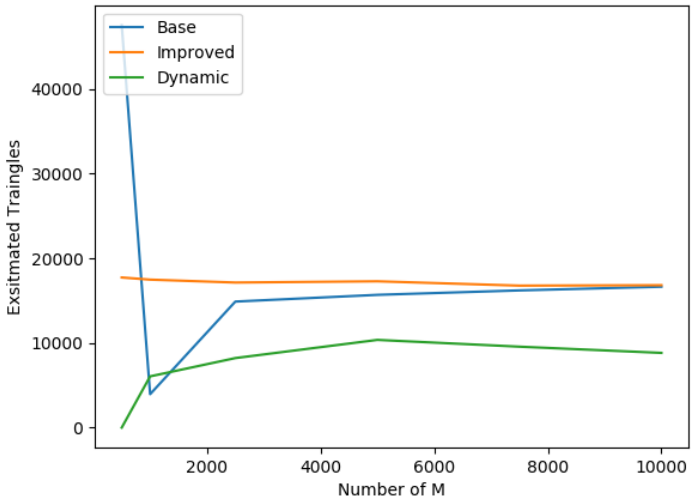

Figure 1: Hamsterster Friendships Triangle Count for various M

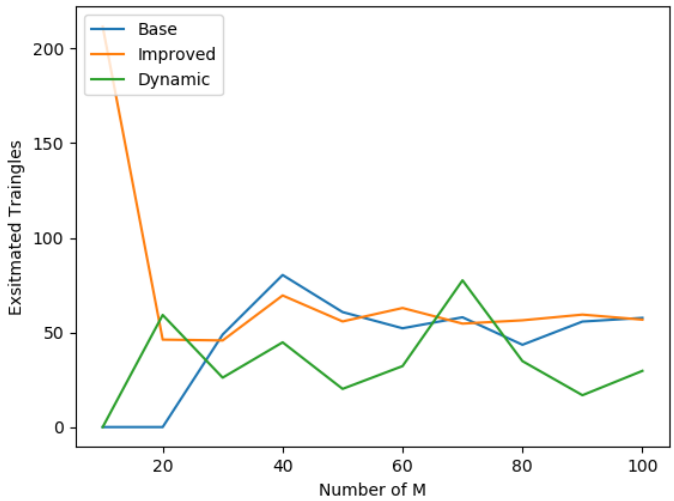

Figure 2: Contiguous USA Triangle Count for various M

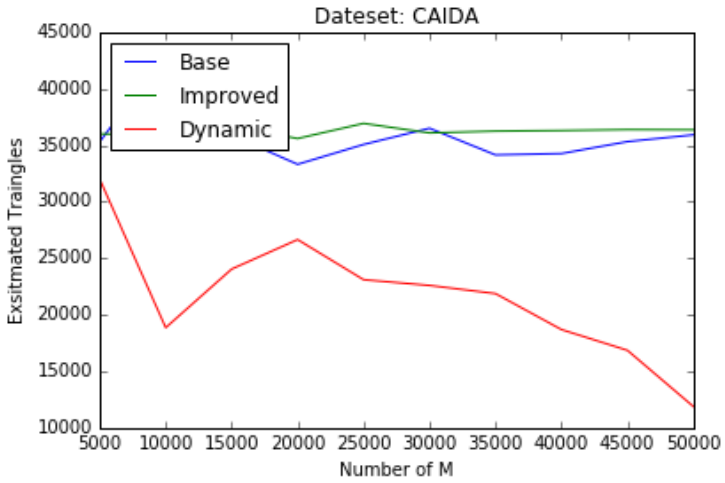

Figure 3: CAIDA Triangle Count for various M

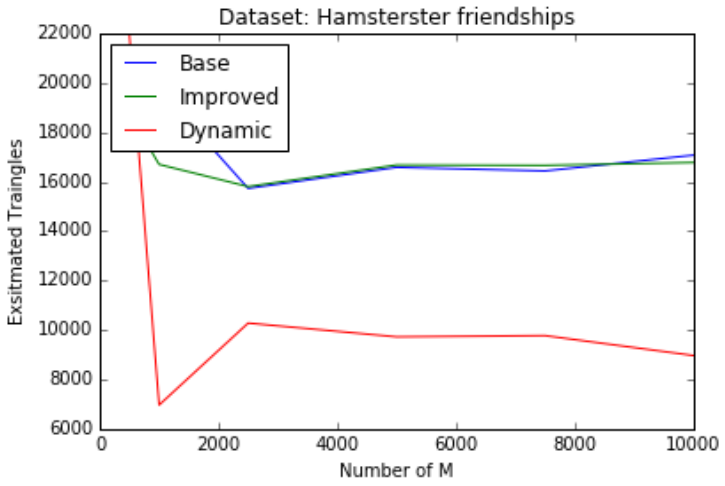

Figure 4: Hamsterster Friendships: Average Triangle Count for various M

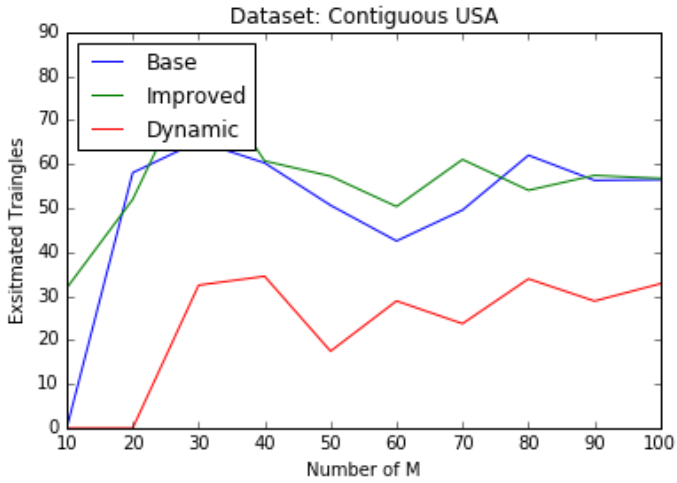

Figure 5: Contiguous USA: Average Triangle Count for various M

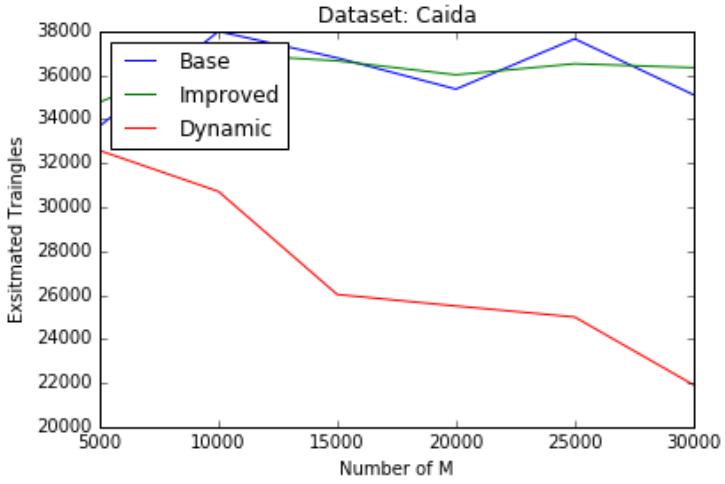

Figure 6: CAIDA: Average Triangle Count for various M

4 Questions for the bonus task

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

  We faced the following challenges:

  1. Each of the team member had a different python version, which

effected how different python functions behaved and resulted in different solutions for the various datasets.

2. We assumed that the python pop() function for sets removes a random element from a list, but it only returned the first one. At a later time in the development, we changed this part of the code to actually remove a random element.

3. We were unsure about the appropriate selection of data sets among the available choices. The data sets selected have similar properties in terms of undirected and unweighted. However, they differ in size.

4. Another main challenge was the debugging of the implementation since it requires t iterations and contains a random factor.

5. We tried to implement the dynamic version in addition to the base and improved version. However, due to missing comparing values for the output, we are unsure about the correctness of its implementation

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

  We think it would be quite difficult to parallelize the algorithm but its not impossible. Since in the algorithm we estimate the global triangles in the graph, we would need information about the neighbourhood of the vertices and edges to update it. We could process different parts of the graphs on multiple servers and estimate the local triangle count there then we need a couple of other serves to maintain communicate between the different graph parts so we can estimate triangles that exist between different graph parts. In addition, a lot of communication will be needed to ensure synchronization of the various counters. To ensure a more accurate global triangle count, we need to use a suitable vertex partitioning algorithm to able to divide the parts of the graphs across multiple serves.

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

  Yes, it can. The main advantage of these algorithms is that they are able to work with unbounded graphs because it uses reservoir method or random pairing for sampling, which means you can define a fixed size of memory you want to utilized and thus a fixed number from the stream. Other related works, which were mentioned in the paper of TRIEST, keep elements by a probability p, which would not work for

unbounded graphs as it would not utilized a fixed memory and would thus grow out of memory and give an error after a while.

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

  Only the fully dynamic version of TRIEST supports edge deletions, since it uses random pairing instead of reservoir sampling and random pairing can handle edge deletion by compensating for them with future insertion of edges. The other two versions: Base and Improved can only handle edge-insertion.