Homework 3 - Mining Data Streams

Perttu Jääskeläinen Group 11 Gabriele Morello
perttuj@kth.se morello@kth.se

November 28, 2022

Contents

1 Introduction

In this homework, we implemented a stream graph processing algorithm for estimating items in large data sets. The choices of algorithms provided for us used either Reservoir Sampling to choose a random set of items from a large population, or the Flajolet-Martin algorithm, which approximates the amount of distinct elements in a stream. We ended up choosing the Trièst [1] paper, which uses Reservoir Sampling along with additional handling for dynamic streams (both insertions and deletions).

The implementation was done in Scala using the Apache Spark framework to process our chosen data set.

2 Algorithm choice - TRIEST

For the algorithm used for our implementation, we chose the Triest [1] paper, which uses Reservoir Sampling to maintain an edge sample S of up to M edges from a data stream.

In the Triest algorithm(s), there are two possible implementations - the basic TRIEST−BASE, with a possible improvement with the TRIEST−IMPR for achieving higher-quality (lower variance) estimations by introducing an improved insertion algorithm - and the TRIEST−FD - or fully dynamic algorithm - for handling streams with both insertions and deletions.

We were given a choice to implement either both TRIEST−BASE and TRIEST−IMPR, or the TRIEST−FD algorithm(s) - where we chose to do the latter.

2.1 Selection of data set

To be able to verify that our algorithm implementation works as intended, we selected a data set from a list of pre-defined sets with known values (nodes, edges and triangles) - so that we would be able to verify if the amount of triangles (total and estimated amounts) from running the algorithm were correct.

The possible data sets were found from the Stanford EDU site, where we chose to use a data set for social circles on Facebook - which is an un-directed graph with ∼4k nodes, ∼88k edges and ∼1.61m triangles. The data set can be found here: https://snap.stanford.edu/data/ego-Facebook.html.

2.2 TRIEST-FD Implementation

The implementation uses regular batch processing with RDD's in Apache Spark, which builds a graph with some specific value for M for the TRIEST algorithm, which determines what size our edge sample graph S will have.

When running the algorithm, we will keep track of certain values as we calculate the result: d_i to keep track of edge deletions where the edge e was in S at the time of deletion, and d_0 for when it *was not*.

2.3 Estimation of triangles

To verify that our implementation worked as intended, we used a smaller size for M than the total amount of edges in our data set (which was ∼88k) - and calculating the approximation using the formula provided in the paper.

Since the data set does not contain any edge deletions, we could more easily calculate the result using the following, at time t:

$$p(t) = \begin{cases} 0 & \text{if } \mathrm{M}^t < 3 \\ \frac{\tau^t}{\kappa^t} * \frac{s^t(s^t-1)(s^t-2)}{M^t(M^t-1)(M^t-2)} & \text{otherw.} \end{cases}$$

For our data set (without deletions), κ^t was always 1 - so we didn't have to take it into account in the calculation. We got the following results when verifying our data sets (file: **edges_and_op.txt**):

- M = 5k, p(t) = 1693536 - approx 5% error
- M = 10k, p(t) = 1580342 - approx 2% error
- M > 88k (entire data set stored in S), p(t) = 1612010, i.e. actual triangle count

Based on these results, we could determine that our implementation was working correctly, and that TRIEST is very good at approximating the total triangle count, even when M is a lot smaller than the actual received data set.

To verify the correctness for edge deletions, we also introduced a random probability for deleting edges in the set, and inserting new ones at a later time. We tested the following:

- 10% deletions, 90% insertions: p(t) = 624721
- 20% deletions, 80% insertions: p(t) = 396868

As with the case for just insertions, the approximation when deleting edges was also very close to the actual triangle count.

3 Bonus part - Questions about the Algorithm

3.1 What were the challenges you faced when implementing the algorithm?

During implementation, we struggled the most with attempting to get the data received to be streamed, rather than processed in batches - as is the usual intention when working with Spark. We ended up going back to just working with RDD's and reading the data set directly, and focused on understanding and implementing the TRIEST algorithm instead, and attempting to convert the data into a "real" stream once done.

The algorithm was fairly straight forward to implement - where most of the time was spent on understanding the individual steps - incrementing the neighborhood triangle counts, the global counts, and how d_i and d_0 were set and how they impacted the final result.

We also had some issues calculating the approximations - since they generated some *very large values* when calculating κ - it was fairly easy to do without deletions, but when d_i and d_0 were greater than 0, calculating κ proved to be fairly difficult.

To solve the problems with large numbers, we had to resort to using Scala's built in **BigDecimal** and **BigInt** types, which are able to store much larger values - up to $2^{2^{64}}$. After some trial and error we were able to calculate very good approximations.

3.2 Can the algorithm be easily parallelized?

We believe that it *can be done* - but it won't be easy. Since insertion and deletion of edges is dependent on the global count of edge sample size S at time t, having multiple servers, for example, receive data from different streams, could result in invalid probabilities since they both might have a copy of t - while it should be t + 1 in one of them.

We believe that it could be parallelized in the following way: we have a centralized server/worker (leader) that handles insertions, deletions, and keeping track of global counters. When receiving new data, workers will request a global time from the leader - which will be used to break ties when multiple parallel servers either want to insert/delete/update counters in the global state.

This way, we can have multiple servers for receiving data from different streams, while maintaining one global state in a leader process. When the algorithm has been running for a long time, assuming we do not have a lot of deletions, the following will most likely fail in each receiving process:

$$FlipBiasedCoin(\frac{M}{t}) = heads$$

Since t will be very large compared to M - in such scenarios, there is no point in having a centralized server that receives *all of the incoming data streams* - if they fail, we do nothing (apart from incrementing the global counter), so it can be parallelized.

3.3 Does the algorithm work for unbounded graph streams?

Yes - the entire point of the algorithm is to maintain an accurate sample of an unbounded stream - only using constant amount of memory, equal to the user specified parameter M.

The only bottleneck we thought of in the algorithm would be if the global counter exceeded the amount possible to store in a single value. For an unsigned long long in c++, this would be 2^{64} - which is most likely more than enough for a stream. If the value is encountered, we can always allocate a new variable - and keep incrementing the count.

Since we're using the Java **BigInt** and **BigDecimal** classes, we will be able to handle values up to $2^{2^{64}}$, which is more than enough for a data stream.

3.4 Does the algorithm support edge deletions?

Yes - it does - and it adapts the approximation algorithm for such cases accordingly. The TRIEST-FD algorithm takes into account both deletions for edges that are in the graph S at time t - and edges that are not.

If an edge is *not in the graph*, the algorithm will discard some future edge to compensate for it. If we've received some amount of deletions for edges that *are in the graph* and also *are not in the graph*, the probability is *the amount of deleted edges from S* divided by *the amount of deleted edges from S + the*

amount of deleted edges not in S. This allows us to discard certain edges, while making sure to fill up the sample size of S up to M eventually.

The deletion counts are represented by d_i, which represents edge deletions where the edge is part of S, and d_0, which represent deletions for edges that are not part of S.

References

[1] M. Riondato L. De Stefani, A. Epasto and E. Upfal. *TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size*. Addison-Wesley Professional, 2016.