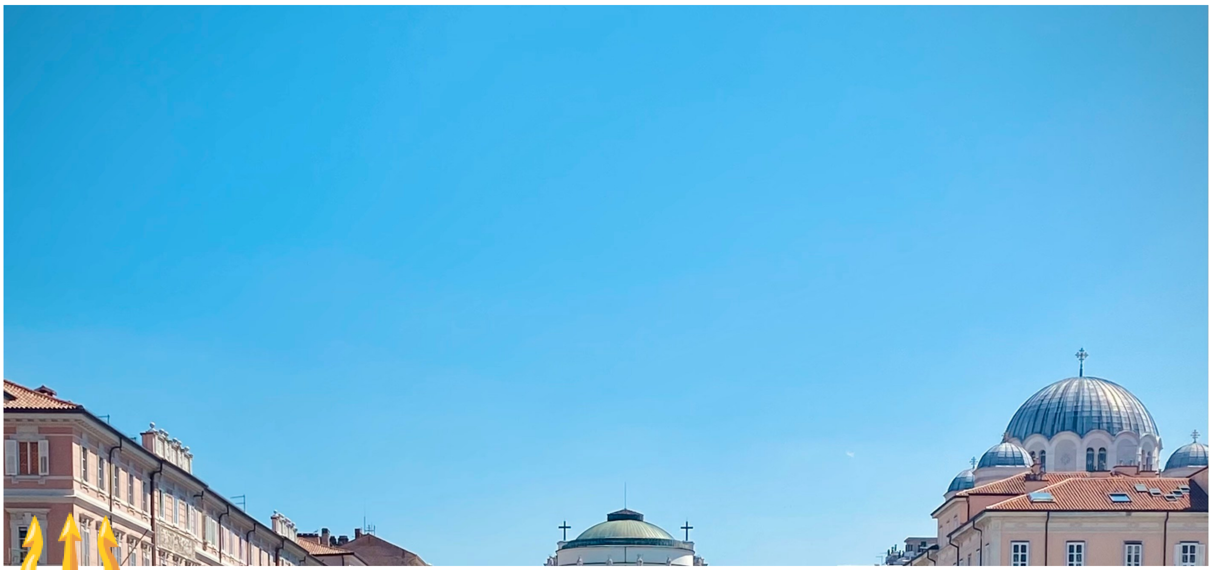

HW3 - Mining Data Streams

💡 This lab is based upon the article "*TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size*" by L. De Stefani et. al., found here.
The algorithms implemented are TRIÉST-BASE and TRIÉST-IMPR.

Solution

The solution is based upon 3 classes, one Main class which streams the data to the algorithms and performs everything not directly linked to the algorithms, and two TRIÉST classes that implement TRIÉST-BASE and TRIÉST-IMPR respectively.

Main

The Main class has one main functionality, which is reading the data stream and passing it on, item by item, to the TRIÉST class of choice. This is done by just looping through the dataset file, formatting the line to a tuple of two values, and calling the *update* method of the chosen TRIÉST algorithm.

After the last entity of the entire dataset, or a specified number of iterations, has been reached, it prints the results of the algorithm and compares it to the true value

(found at the source of the data)

TriestBase

The TriestBase class follows the pseudo-code specified in the paper and implements the algorithm in a very similar way. It has a method, *update*, which takes an edge and samples it, meaning it, depending on the value of t and M, either skips it or adds it to the sample S and in that case updates the counters that give us the global and local number of triangles.

This is done by flipping a biased coin, and if the coin lands on heads, gets the intersection of the neighborhoods of the two vertices of the edge, and then increments the global counter and the local counters for both the vertices in the edge and the ones in the shared neighborhood

The theory behind this is that if two vertices share a node in their respective neighborhoods, they are part of a triangle since the nodes both have to be connected to the third node.

The neighbors for each node in S are stored in a hash table to save computation time, as suggested in the paper.

To get the final estimate of the number of triangles in the dataset, we call the getEstimate-method which multiplies the global counter with a variable ξ, which is a weight dependent on the value of M and t. This is because as we get t:s that are much larger than M, the sample set will be much more sparse, meaning fewer triangles than in the real dataset. We avoid this bias by increasing ξ when t is larger than M.

TriestImpr

The TriestImpr class functions in the same way as the TriestBase class, but with some changes to reduce the high variance of the model. These are that we stop decrementing the counters, and start to always increment them despite the coin flip not showing heads. This is however done with a weight, which is quite similar to ξ.

Now, both the local and the global counters can be found in the counters in the program directly.

This decreases the variance of the model by a lot, while not impacting performance, making it a superior model.

How to run

The code is simply run by running the main.py class. To change the model used, change the “algorithm” parameter when creating the main class to either “Base” for TRIÉST-BASE or “Impr” for TRIÉST-IMPR.

Data

The dataset used to test the implementation is the Stanford Web Graph, found here

Results

TRIÉST-BASE

M = 20000

Output from running the main file with n = 10, M = 20000 using the TRIÉST-BASE algorithm

```
Results of running 10 tests with M = 20000 and algorithm = TriestBase
    -Test 1 estimate: 8902273 difference: -2427200 (-21.424%) time: 60 seconds
    -Test 2 estimate: 12858839 difference: 1529366 (13.499%) time: 58 seconds
    -Test 3 estimate: 7913132 difference: -3416341 (-30.154%) time: 58 seconds
    -Test 4 estimate: 14837122 difference: 3507649 (30.96%) time: 59 seconds
    -Test 5 estimate: 12858839 difference: 1529366 (13.499%) time: 58 seconds
    -Test 6 estimate: 7913132 difference: -3416341 (-30.154%) time: 58 seconds
    -Test 7 estimate: 10880556 difference: -448917 (-3.962%) time: 58 seconds
    -Test 8 estimate: 16815405 difference: 5485932 (48.422%) time: 59 seconds
    -Test 9 estimate: 3956566 difference: -7372907 (-65.077%) time: 58 seconds
    -Test 10 estimate: 9891415 difference: -1438058 (-12.693%) time: 58 seconds
```

M = 50000

```
Results of running 5 tests with M = 50000 and algorithm = TriestBase
    -Test 1 estimate: 11520483 difference: 191010 (1.686%) time: 285 seconds
    -Test 2 estimate: 11330585 difference: 1112 (0.01%) time: 266 seconds
    -Test 3 estimate: 12406674 difference: 1077201 (9.508%) time: 272 seconds
    -Test 4 estimate: 10001299 difference: -1328174 (-11.723%) time: 272 seconds
    -Test 5 estimate: 12723171 difference: 1393698 (12.302%) time: 265 seconds
```

TRIÉST-IMPR

M = 20000

Output from running the main file with n = 10, M = 20000 using the TRIÉST-IMPR algorithm

```
Results of running 10 tests with M = 50000 and algorithm = TriestImpr
    -Test 1 estimate: 13054483 difference: 1725010 (15.226%) time: 62 seconds
    -Test 2 estimate: 10641922 difference: -687551 (-6.069%) time: 61 seconds
    -Test 3 estimate: 13458574 difference: 2129101 (18.793%) time: 61 seconds
    -Test 4 estimate: 10698864 difference: -630609 (-5.566%) time: 63 seconds
    -Test 5 estimate: 10474490 difference: -854983 (-7.547%) time: 61 seconds
    -Test 6 estimate: 10797105 difference: -532368 (-4.699%) time: 62 seconds
    -Test 7 estimate: 11778596 difference: 449123 (3.964%) time: 62 seconds
    -Test 8 estimate: 11348349 difference: 18876 (0.167%) time: 62 seconds
    -Test 9 estimate: 11194148 difference: -135325 (-1.194%) time: 61 seconds
    -Test 10 estimate: 13507621 difference: 2178148 (19.226%) time: 60 seconds
```

M = 50000

```
Results of running 5 tests with M = 50000 and algorithm = TriestImpr
    -Test 1 estimate: 10910320 difference: -419153 (-3.7%) time: 292 seconds
    -Test 2 estimate: 10917708 difference: -411765 (-3.634%) time: 275 seconds
    -Test 3 estimate: 12059671 difference: 730198 (6.445%) time: 275 seconds
    -Test 4 estimate: 11461011 difference: 131538 (1.161%) time: 272 seconds
    -Test 5 estimate: 11690863 difference: 361390 (3.19%) time: 271 seconds
```

Bonus

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

- With so many data points that have to be sifted through, it was sometimes hard to debug using the debugger. Also, the fact that there is randomness in the algorithm makes it much harder to corner bugs and problems in order to find out why they appear.
- Since the data set is directional, I had some problems with how that should be handled. Sometimes the same node appears twice, which put me in a spot where I had to search how to define a triangle in a directed graph. In the end, I implemented it in a way that just ignores the fact that they are directional, but that created a problem where the same graph can appear several times. I solved this by using a set for S, meaning there can be no duplicates.

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

- Yes!
- If we split the datastream into two streams (of length n and m respectively), and perform the algorithm on them separately (and in parallel), we can later do another similar pass over both of them.
- In this second pass, the subsets $S_{1,2}$ generated by the parallel algorithms will both be M long. We can now pass through them and for every iteration pick the corresponding item from S_1 with probability $p = \frac{n}{n+m}$, and otherwise pick the corresponding item from S_2. The probability is to account for the different lengths of the streams since the longer stream has more data behind it and should hence be weighted higher.

https://ballsandbins.wordpress.com/2014/04/13/distributedparallel-reservoir-sampling/

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

- Yes!
- As long as the graph is linear, it can be unbounded. More data can be added to it to it indefinitely. This is because the algorithm does not need to know anything else than the next edge in order to compute the estimate.

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

- No, it does not.
- To fix this, we could implement the “random pairing”, which basically keeps track of how many deletions are not made (because the edge to be deleted may not be in S). An intuitive way to think about this is that if we miss a deletion, we could skip an insertion to compensate. With the premise of each edge being equally likely to be in a triangle, this should even out in the long run.