ID2222 Data Mining Homework 3

Mining Data Streams

Luís Santos
lmpss@kth.se

Kevin Dalla Torre
kevindt@kth.se

November 2020

1 Paper selection

We have opted for the 2nd paper in the homework description. *TRIÈST* (De Stefani et al., 2016) presents a suite of fixed memory size algorithms to estimate local/global triangle counts in a streaming graph consisting of an ongoing stream of edge insertions/deletions. We have selected TRIEST-BASE and TRIEST-IMPR algorithms for implementation, which deal with the insertion-only case. We shall now discuss our solution in more detail.

1.1 Reservoir sampling

We implemented the reservoir sampling technique in the *sample_edge* method. In the base case, when $t \leq M$, the edge gets deterministically inserted into the subgraph $\mathcal{S}$. If $t > M$, we flip a biased coin with heads probability M/t, and on the event of *heads* we sample a random edge from $\mathcal{S}$ with an uniform distribution. We then remove the sampled edge from our subgraph, and in the case of TRIEST-BASE we decrement the counters for the sampled edge. The method *sample_edge* returns True if the edge e_t under consideration is to be inserted into the subgraph, and False otherwise.

1.2 Updating the counters

In both TRIÈST implementations, we maintain one global counter and a set of local counters for a subset of the nodes. These counters are used to

estimate the global and local number of triangles respectively. For an edge $e_t = (u, v)$, the method *update_counters* obtains the neighbors of node u and neighbors of node v in subgraph $\mathcal{S}$, and computes a shared neighborhood of u and v in $\mathcal{S}$. For each shared neighbor c, we update the global and local counters of u, v and c. Depending on the selected operator $+$ or $-$, it can be an incrementing or decrementing operation. We make sure to delete the local counters which have reached a value of 0, in order to retain a desirable memory footprint. For TRIEST-IMPR, *update_counters* deals only with $+$ operators, and we compute the value η_t to increment the counters with in accordance with the following formula: $\eta_t = \max\{1, (t-1)(t-2)/M(M-1)\}$.

1.3 The subgraph $\mathcal{S}$

The subgraph $\mathcal{S}$ is an important auxiliary data structure for this problem. In our solution, we opted for an adjacency list data structure using dictionaries. By explicitly storing the neighbors of each node u, it makes it efficient to compute the shared neighborhood of any pair of nodes u and v. To preserve the memory constraints, we make sure to remove nodes from the adjacency list if their degree is 0. At any point in time, we have at maximum M edges present in the adjacency list given the properties of reservoir sampling.

1.4 Algorithm and estimation

The method *algorithm* computes the outer-level algorithm. We initialize an empty subgraph and the counters at 0. For each edge in the insertion stream, we call *sample_edge* method which tells us if the current edge at time t is to added to the subgraph, after the potential removal of a random edge to make space for the new one. In case it is to be added, we add this edge to the subgraph $\mathcal{S}$ and for TRIEST-BASE we update the counters for that edge. For TRIEST-IMPR, we update the counters before the if statement, regardless of whether the current edge is to be kept or dropped. For TRIEST-BASE, we compute $\xi_t = \max\{1, t(t-1)(t-2)/M(M-1)(M-2)\}$ for the estimation. To estimate the triangle counts, we multiply ξ_t with the global/local counters to obtain the global/local triangle estimations. Whereas for TRIEST-IMPR, the counters themselves provide the unbiased estimate of number of triangles.

2 Bonus tasks

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

The authors mention on Section 2 that TRIEST algorithms are specific for undirected graphs. We ran into some problems when evaluating our solution for directed graphs such as the case of the snap web graphs. We fixed these problems by attempting to convert a directed graph to an undirected graph. As we process an edge (u, v), we invert it to (v, u) if $u > v$ to guarantee that u is smaller than v. We also remove self-referencing nodes with edges which point to themselves. That means we discard edges if $u = v$. In our adjacency list for an edge (u, v) we always retain a bidirectional connection from u to v and v to u.

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

It might be possible to parallelize the algorithm, but it would require substantial algorithmic changes and would require synchronization between workers. The main issue is that the algorithm depends on a global counter for a global triangle estimate, as well as local counters for local estimates. The computation of global and local counters requires information about the neighbors of nodes pertaining to the edge being processed. A parallel execution would thus require synchronization among workers to update the counters taking into account the current graph partitions, making this a difficult algorithm to parallelize.

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

The algorithm does work with unbounded graph streams. Due to reservoir sampling, the algorithm maintains a fixed memory footprint M. With a fixed memory space M we avoid the common pitfalls of memory under-utilization or running out of memory, thus ensuring the algorithm works for unbounded streams and uses the available space efficiently. This is in contrast to edge sampling schemes which maintain a fixed edge probability p that can easily encounter the under- or over-utilization problems just described.

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

The implemented algorithms are insertion only algorithms. However, in the paper they have another algorithm called Fully dynamic algorithm that handles both insertions and deletions which is based on random paring (RP). The idea is that edge deletions will be compensated by further edge insertions seen later in the stream. The way it keeps track is by having counters to keep track of uncompensated edge deletions.

3 How to run

Our homework solution can be run via the command `python run_hw.py`. Check Figure 1 for the command-line parameters, as well as a short description of the functionality of each parameter. Check Figure 2 to observe an example run, as well as to obtain the default values of the parameters. We used the sales transaction dataset as provided to us in the homework statement. To run our Python script, call the script `run_hw.py` with the optional command line arguments.

```
(data-mining) luis@luis-PC:~/kth/data mining/id2222-data-mining/homework 3$ python run_hw.py --help
usage: run_hw.py [-h] [-dataset-file DATASET_FILE] [-triest {base,impr}]
                 [-M M] [-verbose]

Find global/local triangle estimates in a streaming graph using TRIEST.

optional arguments:
  -h, --help            show this help message and exit
  -dataset-file DATASET_FILE
                        path to a gzipped snap dataset
  -triest {base,impr}   TRIEST algorithm
  -M M                  size of the sample of edges used in reservoir sampling
  -verbose              decides if the results are printed
(data-mining) luis@luis-PC:~/kth/data mining/id2222-data-mining/homework 3$
```

Figure 1: Command-line parameters of our solution.

```
(data-mining) luis@luis-PC:~/kth/data mining/id2222-data-mining/homework 3$ python run_hw.py
Namespace(M=10000, dataset_file='web-Stanford.txt.gz', triest='impr', verbose=False)
M: 10000, dataset_name: web-Stanford.txt.gz
Global triangles estimate: 16929304
TRIEST-IMPR took 23.403s
```

Figure 2: Default values of command-line parameters for an example run.

References

L. De Stefani, A. Epasto, M. Riondato, and E. Upfal. TRIÈST: Counting local and global triangles in fully-dynamic streams with fixed memory size. pages 825–834, 08 2016.