ID2222 Data Mining Homework 3

GIOVANNI MANFREDI SEBASTIANO MENEGHIN
gioman | meneghin @kth.se

27th November 2023

1 Introduction

Our project consists of the full tasks and questions presented in the project description, including the optional questions for the possible additional two points. We have create a set of classes able to perform both Triest Base and Triest Improved on the provided dataset.

As first step we enter the data from a *.gz* file present in our project folder and then we process those data. Then the selected data, according to some parameter that can be inserted by the users, pass through the procedure described above and the final results are the estimates of the present triangles.

The timing results are presented in the subsection 2.3, where the *project_executor.py* is called both running Triest Base and Triest Improved algorithm.

2 Task 1: Triest Implementations

2.1 Data extraction and processing

The dataset used in this project is the Stanford web graph, available on Stanford Website. This represent a dataset collected in 2002 of pages from Stanford University website (as nodes) and the represent hyperlinks between them (as directed edges). It contains a total of 281903 nodes and 2312497 edges, that together form 11329473 triangles. The nodes and the edges are taken from the *.zp* file and the graph is logically created, thanks to adjacency lists and edge set.

2.2 Algorithm explanation

The Triest Algorithms efficiently estimates the number of triangles in a streaming graph by leveraging reservoir sampling techniques. The *algorithms* method orchestrates the entire process, both for Triest Base and for Triest Improved, making use of other methods to handle specific tasks within the algorithms. The algorithms utilize reservoir sampling techniques to efficiently process a continuous flow of graph edges while maintaining a fixed maximum number of considered edges (M).

The *algorithm* code start with the access an edge stream taken from the dataset file and the initialization of $t = 0$. For each edge (u,v) between the nodes (u,v), update the local and global counters, each initialized to 0 during the creation of the class *TriestBase* or *TriestImpr*. If we are in the **TriestBase**, the counters are **updated according unitarily**, **whereas in TriestImpr** the counters are updated **according to** the online computed **value of eta**, given t.

Then, this edge is added to the subgraph created by the starting graph, if $t ¡ M$ or if with probability M/t the script decides to remove another graph from the adjacency lists and edge sets and add this new edge (u,v). If this it is added, the adjacency lists and the edge set are updated.

Lastly, the global triangles' esteem is printed, by retrieving the value of the global counter. If the *flag verbose* is set to true when the script is called by terminal, also the information about the local

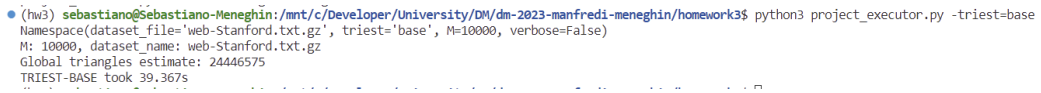

Figure 1: Timings for Triest Base

Figure 2: Timings for Triest Improved

triangles' esteems are printed. In the case of the **TriestBase**, the **value of the all these esteems will be moderated by** the value of **eta**, calculated beforehand.

2.3 Results

There is a single runnable file, that can be used to launch both the algorithms by changing a default value, present into the project folder with the name of *project_executor.py*. Both the algorithms have been tested on a WSL subsystem built on Windows 11 running on a laptop with 16GB RAM and CPU Intel i7 series 12.

The Triest Base algorithm, with the default value M = 10000, performed as shown in Figure 1

Instead for Triest Improved algorithms, still tested on the same datasets with M = 10000, has computed with a slighlty lower time, as shown by the results in Figure 2.

It has to be noticed that the shown results might not be the same result you will find running the project, due to the randomness inserted in the project by the function *simple_edge*, both in Triest Base and in Triest Improved. Furthermore, you might have different estimates, with different timing needed for the total computation. This last thing can change also depending on which is the machine you are running the code in.

2.4 How to run

In order to run the project, you must have installed some packages in your own python environment, as gzip. What is needed can be found in *requirements.text*, that can actually used to install the through the pip command. We have used Python 3.10.12 for this specific tasks and we suggest to use Python 3.8.0 or an higher version, since some commands of some primitives and libraries have been changed from older versions' ones.

The full project can easily be run via terminal using the command *python3 project_executor.py* and the timings will be shown in any case without any other additional command. The dataset is provided due to recent problems with the SNAP Website. However, it can be independently downloaded from the source provided as described in subsection 2.1. Once you have downloaded the *.gz* file and placed it in a folder called *datasets*, your program will recognize the provided files and then process them (or you can change the path on your copy of the code in the file *data_extractor.py*, if this solution works better for you).

3 Task 2: Optional Questions

3.1 Question 1: What challenges did you encounter during the algorithm implementation?

The authors highlight in Section 2 of their paper that TRIEST algorithms are specifically designed for undirected graphs. We faced issues when applying our solution to directed graphs, such as the snap

web graphs. To address this, we resolved to convert a directed graph to an undirected one. When processing an edge (u, v), we inverted it to (v, u) if u ¿ v to ensure u is smaller than v. Additionally, we eliminated self-referencing nodes with edges pointing to themselves, discarding edges where u = v. In our adjacency list for an edge (u, v), we consistently maintained bidirectional connections from u to v and v to u.

3.2 Question 2: Is the algorithm easily parallelizable? If so, how? If not, why? Explain.

While it may be possible to parallelize the algorithm, significant algorithmic changes and synchronization between workers would be required. The primary challenge lies in the algorithm's dependence on a global counter for a global triangle estimate, along with local counters for local estimates. Computation of these counters necessitates information about the neighbors of nodes related to the processed edge. Achieving parallel execution would demand synchronization among workers to update counters, accounting for the current graph partitions, making parallelization challenging.

3.3 Question 3: Does the algorithm function with unbounded graph streams? Explain.

Yes, the algorithm is compatible with unbounded graph streams. Leveraging reservoir sampling, the algorithm maintains a fixed memory footprint denoted as M. This fixed memory space mitigates issues of memory under-utilization or exhaustion, ensuring the algorithm efficiently handles unbounded streams and optimally utilizes the available space. This sets it apart from edge sampling schemes with fixed edge probability p, which are susceptible to under- or over-utilization problems.

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

The implemented algorithms are designed for insertion-only scenarios. However, the paper introduces another algorithm, the Fully dynamic algorithm, capable of handling both insertions and deletions based on random pairing (RP). The concept is that edge deletions are compensated by subsequent edge insertions in the stream. Counters are employed to keep track of uncompensated edge deletions in this dynamic algorithm.