ID2222 Assignment 3 - Mining Data Streams TRIEST - Counting Global Triangles - Javier-G Hamed-M

November 26, 2017

```
In [1]: %reload_ext watermark
        %watermark -a 'Antonio Javier González Ferrer & Hamed Mohammadpour' -v -d -r

Antonio Javier González Ferrer & Hamed Mohammadpour 2017-11-26

CPython 3.6.3
IPython 6.1.0
Git repo: git@github.com:jgonzalezferrer/triest.git
```

1 Introduction

In this notebook, we will study ways to analyze data streams, more explicitly Sampling from streams using **Reservoir** method and *counting global and local triangles in a fully-dynamic undirected graph* as described in this paper called **TRIÉST**:

1.1 Algorithms

The paper presents three algorithms for counting local and global count of triangles, and we implemented and tested two following algorithms:

1. TRIÉST Base
2. TRIÉST Improved

1.1.1 Fast overview

1. Importing required modules We created custom classes for each algorithm in the paper in addition to `Graph` and `EdgeStream` classes.

```
In [2]: from triest.algorithms import TriestBase, TriestImpr
        from triest.graph import Graph

        from triest.stream_graph import EdgeStream

        from tqdm import tqdm, trange, tqdm_notebook # Printing progress bar
```

2. Reading data and building the graph We start by reading our data file which is consist of edges in a graph; we use our `Graph` module to load it as its instance.

```
In [3]: file = 'data/out.arenas-jazz'
        graph = Graph.open_file_as_graph(file)
        edge_stream = EdgeStream(graph.edges)
```

3. Initilization We define our desired value for hyper-parameter `M` in the algorithm and create an instance of the `TriestBase` class to run it later on.

```
In [4]: M = 2500
        tb = TriestBase(edge_stream.elements, M)
```

4. Running TRIÉST-Base case and Results By running our `TriestBase` instance, we get an estimation of the number of triangles as follow on our test data:

```
In [5]: %%time

        tb.run()

        print("'t' is: {:d}".format((tb.t)))
        print("Estimated number of triangles is: {:d}".format(tb.triangles_estimation()))
```

```
't' is: 2742
Estimated number of triangles is: 13692
CPU times: user 1.19 s, sys: 26.2 ms, total: 1.21 s
Wall time: 1.26 s
```

5. Running TRIÉST-IMPR and Results In the next section, we analyze the performance of the *TRIÉST-IMPR*:

```
In [6]: %%time

        ti = TriestImpr(edge_stream.elements, M)
        ti.run()

        print("Estimated number of triangles is: {:d}".format(ti.triangles_estimation()))
```

```
Estimated number of triangles is: 17596
CPU times: user 1.04 s, sys: 26 ms, total: 1.07 s
Wall time: 1.1 s
```

1.2 Bonus Questions

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

- Benchmarking against count the of local triangles became ambiguous as we don't have any data from our dataset.

1.2.2 Can the algorithm be easily parallelized? If yes, how? If not, why?

The algorithm can be parallelized if we can modify the counting neighbors method with one of the parallel implementation of it. Doing so, as the algorithm based on one global variable to count the triangles, we should select one of parallelization paradigms, such as **"Parameter Server - Worker"** approach or other ones to update the counter as workers produce results.

But all of these mean that this algorithm is not easily parallelizable.

1.2.3 Does the algorithm work for unbounded graph streams?

Yes, the main advantage of this family of algorithms to other related works mentioned in section 3 is being able to process unbounded graphs as it uses *Reservoir method* for sampling which sample has a fixed number from the stream, fully utilizing the available memory. The other approach of keeping elements by a probability p wouldn't work for unbounded graphs as it would under-utilize memory at first and as time goes, the memory utilization will grow until an `Out of Memory` error.

1.2.4 Does the algorithm support edge deletions? If not, what modification would it need?

The first two versions of the algorithm support only *'edge-insertion '* while the **Fully-Dynamic** version can handle edge insertion and deletion.

For making the algorithm parallel in Fully Dynamic version, it uses **Random Pairing (RP)** rather than *Reservoir sampling*. RP extends the latter to handle edge deletions by compensating them with future edge insertion.