Lab3 - In-Core Computation of Geometric Centralities with HyperBall
Data Mining - KTH

Andrea Scotti, Daniele Montesi

26 November 2019

1 Introduction

The goal of the assignment is to implement an algorithm in one of the three paper presented in the Lab description. In this occasion, we will present **HyperBall**, an algorithm for node centrality computation that relies on the use of approximative counters (or cardinality approximators) for big graphs. As first step towards the task completion, we implemented the algorithm for the **Flajolet-Martin** presented in the lectures. As second step, we implemented the HyperBall Algorithm. The Dataset we used is a graph of the airports connections in the US, containinc circa 500 nodes and 2000 edges. It is an undirected graph.

Note: The paper uses a modified version of this counter under the name of **HyperLogLog**. For this purpose, we implemented additionally the HyperLogLog algorithm that we then used for HyperBall.

2 Code Explanation

The code is organized in a python project with 5 packages

1. **cardinality**: contains all the files belonging to the cardinality calculator. (Base class: CardinalityCalculator)

2. **centrality**: contains all the files belonging to the centrality calculator. (Base class: centralityCalculator)

3. **graph_dir**: contains all the files belonging to the graph and their elements. (Base class: Graph)

4. **hashers**: contains all the files belonging to the hashers, providing standard random hash functions.

5. **testers**: package containing all the testers scripts for the algorithm

The the lab tasks are here explained.

1. **Flajolet-Martin**: The algorithm is implemented in the file flajoletMartin.py, and can be initialized defining a set of elements (streams) of whom the cardinality should be computed. To return the approximate cardinality, it is possible to call the **calculate()** method.

2. **HyperLogLog**: The cardinality calculator that HyperBall makes use of. Here we report a description of our implementation. HyperLogLog needs 2 inputs:

   - **max_value**, maximum element value allowed
   - **num_buckets** i.e. the number of bits used by the algorithm in order to work and perform the count. (also called M).
   - **hasher**, to provide an hash value for the elements

   The algorithm initializes M registers to zero as a list: [0, 0, 0, ... 0] (size of M zeros). Those are the registers that will be filled and provide the final cardinality. They can be filled by the method union() given another counter, or by the count() whenever a new number is encountered.
   The main idea of HyperLogLog is to keep track of the least significant bit of a given element and do a resoning over the buckets (i.e. the list of zeros). For every number, the LogLog counter returns the **index** of the bucket_list where the element will be stored. For every number we encounter, we will evaluate the update by keeping only the **max** between the 2 values:
   $$max(bucket[bucket\_id], new\_value)$$
   . **Oss**: the new value is determined by the least significant bit of the hashed value of element. Eventually, we will convert the buckets list into a cardinality calling the function **count()**.

3. **HyperBall**: the algorithm of the paper works by computing all the possible cardinalities of each node of the graph, calculated at every step (or walk) performed by starting on that node. To do so, we start iteratively with step t=0, when all the nodes have cardinality 1 (only the node itself), and then updates following a dynamic programming fashion, until convergence. The algorithm works in 2 steps:

   (a) Initialize all the data structures to be used. in particular we need to initialize:

      - **hyper_ball_cardinalities**: the cardinality of all the nodes at every instant t of the computation. Initially is an empty list, it will then become a matrix of size $num\_nodes \times T$ (last step until convergence).

- **hyperloglog_balls_diff**: List of size *num_nodes* filled with tuples. Each tuple contains 2 HyperLogLog counters: the first belonging to step t, the second belonging to step t+1. The list is iteratively updated by the algorithm.
- **hyperloglog_balls_cumulative**: the list of size num_nodes containing the counters of the current step for each node.

(b) After the initialization, we can call **calculate_hyper_balls()** method, which is in charge of computing the cardinalities. The trick is that, instead of calculating the cardinalities adding element by element of newly reached node, we exploit the property:

$$\begin{aligned}\mathscr{B}_G(x,0) &= \{\, x \,\}\\ \mathscr{B}_G(x,r+1) &= \bigcup_{x\to y} \mathscr{B}_G(y,r) \cup \{\, x \,\}.\end{aligned}$$

Figure 1: Cardinality property taken from the study paper.

We perform the union above calculating the max over the current node counter's buckets, and the the neighbor nodes counters' buckets. That is not surprising: as we discussed during the HyperLogLog section, the cardinality is "kept" as a list of buckets that are updated keeping the max value of each bucket, and that's what happens with HyperBalls calling the method **union()** on the counters.

The algorithm will eventually stop when eached convergence (none of the counters saved in hyper_ball_cardinalities changes from the previous step calculation).

In the package **tester**, you can find all the tests that we built in order to check to correctness of the algorithms. In particular we compare the exact cardinalities and centralities with the approximate ones.
The exact cardinality of a multiset is computed by calling the method *len*() on the set where the multiset is stored, in the class *CardinalityTrueCalculator*.
The exact centralities are computed by running a BFS from each node in the class *CentralityTrueCalculator*

3 How to Run the Code

To run the code follow the steps above:

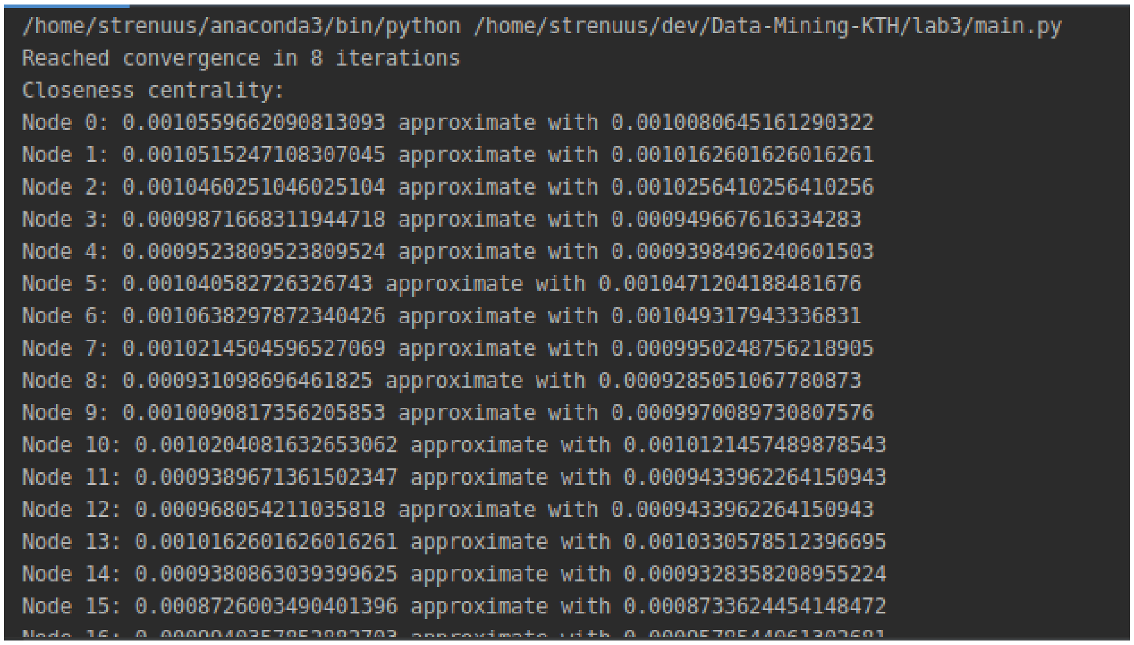
Requirements: Python 3.6, PyCharm (or a similar IDE for Python)

1. open your IDE and

2. navigate inside the directory tester
3. click on python main.py then press run (alternatively, type in the terminal):

   ```
   > python main.py
   ```

4. you should see the following output:



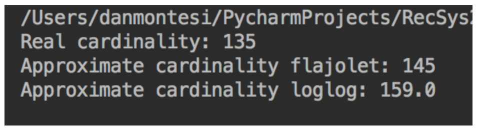
```
/home/strenuus/anaconda3/bin/python /home/strenuus/dev/Data-Mining-KTH/lab3/main.py
Reached convergence in 8 iterations
Closeness centrality:
Node 0: 0.0010559662090813093 approximate with 0.0010080645161290322
Node 1: 0.0010515247108307045 approximate with 0.0010162601626016261
Node 2: 0.0010460251046025104 approximate with 0.0010256410256410256
Node 3: 0.0009871668311944718 approximate with 0.000949667616334283
Node 4: 0.0009523809523809524 approximate with 0.0009398496240601503
Node 5: 0.001040582726326743 approximate with 0.0010471204188481676
Node 6: 0.0010638297872340426 approximate with 0.001049317943336831
Node 7: 0.0010214504596527069 approximate with 0.0009950248756218905
Node 8: 0.000931098696461825 approximate with 0.000928505106778087 3
Node 9: 0.0010090817356205853 approximate with 0.0009970089730807576
Node 10: 0.0010204081632653062 approximate with 0.0010121457489878543
Node 11: 0.0009389671361502347 approximate with 0.0009433962264150943
Node 12: 0.000968054211035818 approximate with 0.0009433962264150943
Node 13: 0.0010162601626016261 approximate with 0.0010330578512396695
Node 14: 0.0009380863039399625 approximate with 0.0009328358208955224
Node 15: 0.0008726003490401396 approximate with 0.0008733624454148472
```

Figure 2: For each centrality measure and for each node, it is possible to compare the exact and the approximate value.

If you want to see the output for the cardinality calculator only, you can run the file:

```
> python cardinalityCalculator.py
```

You should be able to see the following output:



```
/Users/danmontesi/PycharmProjects/RecSys
Real cardinality: 135
Approximate cardinality flajolet: 145
Approximate cardinality loglog: 159.0
```

Figure 3: Output of cardinality tester. Note: every run results different as the algorithm creates random hashes and different data!

4 Additional questions

1. **What were the challenges you have faced when implementing the algorithm?**
   HyperBall algorithm, contrarily to what stated in the lab assignment, do not use the Flajolet-Martin and neither is possible to adapt Flajolet-Martin for the use of the algorithm. Instead, the paper uses a modified version of this counter under the name of HyperLogLog. For this purpose, we needed to implement additionally the HyperLogLog algorithm that then we then used for HyperBall and study how the 2 algorithm worked together.

2. **Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.**
   Yes, similarly to how happens in some library for scalable graph computation, such as PowerGraph and Pregel, we can instantiate the vertex-parallelization of the graph for every iteration t. At each current step t, the results of the previous step t-1 can be available to a shared memory (PowerGraph) or on the driver.

3. **Does the algorithm work for unbounded graph streams? Explain.**
   With unbounded we intend a type of ever-growing, essentially infinite data set. The algorithm for Flajolet-Martin can easily capture the growth (being it built to compute the cardinality for streaming data). Instead, if we consider an ever-growing graph dataset, using HyperBall we should re-compute all the cardinalities from scratch.

4. **Does the algorithm support edge deletions? If not, what modification would it need? Explain.**
   We are considering the HyperBall algorithm here since we are talking about graph data. In the case of edge-removal, Hyperball needs to re-compute all the cardinalities from instant t=1 without considering the removed edge. That is because the cardinalities calculated are approximations. Hence, the algorithm must be rerun. A possible modification to avoid such could be saving all the possible combinations of edges coming from a particular neighbor, however, this results highly expensive in memory and badly scalable, it's hence not possible to allow edge removal from the dataset using HyperBall.