Homework 3 Report – Mining Data Streams

In Core Computation of Geometric Centralities with HyperBall: a Hundread Billion Nodes and Beyond

Data Mining – ID2222 – Homework Group 23

Adriano Mundo (mundo@kth.se), Edoardo Realini (erealini@kth.se)

November 23, 2020

1. Introduction

The goal of the *homework assignment 3* was to implement one of the three papers that take advantage of the algorithms that we have seen in the theory lectures [1]. All the papers present graph processing methods and algorithms that makes use of stream mining algorithms. We decided to implement the paper n.3: In-Core Computation of Geometric Centralities with HyperBall: a Hundread Billion Nodes and Beyond [2], which leverages on the *Flajolet-Martin* algorithm (HyperLogLog counters) in the original paper since 1985 [4]. Thus, to complete the assignment we implemented both the *Flajolet-Martin* algorithm and the *HyperLogLog* algorithms for approximately counting distinct elements and the *HyperBall* algorithms that leverages on the *HyperLogLog*; where a little description is presented in [2], but the main reference for the complete implementation was the original paper by Flajolet [5].
The following report explains the implementation and how to run the code for the two tasks described above. The implementation has been done by using Python and we leveraged on the knowledge from the book [3] and the citated scientific papers [2,4,5].

2. Implementation

The code is organized in classes, most of the code files attached with this report represents a class. For each task of the assignment there is a class or multiple classes with the respective methods. They are used for running the *Hyperball* algorithm in the *main.py* file. In this section we will explain what each file contains and what is used for.

- ***graph package:*** this package contains the implementation in form of classes of a basic graph structure. The implemented classes are: *Node, Edge* and *Graph*. This class will be used to create graph objects and to work on these graphs later on in the proper algorithm *Hyperball*.

- ***graph_reader:*** this python file contains the implementation of a text file parser in form of class. The input text file the rows are of the format: *(start_node_id, end_node_id, weight)* representing all the edges. The method *read_graph* parses each line ad fills an object of class *Graph* with all the information. Please note that this parser only works with such formatted text files. To use different starting datasets an ad-hoc parser should be implemented.

- ***CentralityHyperBall:*** this file contains the implementation of the Hyperball algorithm presented in [2]. This class makes use of the *NumPy* library and the class defined by us: *HyperLogLog*, which contains the implementation of the counters.

  In *CentralityHyperball* the main attributes are:
  - The graph that is given when constructing the object. The graph must be of class *Graph*.
  - The matrix of hyperball cardinalities which contains in each cell the cardinality of the ball *B(x,r)* where x is the node (line index) and r is the radius of the ball (column index). This matrix is filled by the method *compute_hyper_balls.*
  - *hyperloglog_balls_cumulative*, which contains the cardinality of a node at iteration t
  - *hyperloglog_balls_diff,* a list that registers the counters of a node at instant t and t + 1 respectively in positions 0 and 1 in sub-lists.

  The main methods:
  - *init:* initializes all the attributes described above with the iteration t = 1 of the algorithms.
  - *compute_hyper_balls*: computes all the hyper ball cardinalities and fills the matrix defined in *init* as *hyper_ball_cardinalities*
  - *has_reached_convergence*: method that checks at each iteration if the hyperball algorithm has reached the condition of convergence that is: if no counter has been updated in the last iteration (in our tests normally the iterations are 8 with respect to 500 nodes).
  - The remaining 3 methods: *calculate_closeness, calculate_lin and calculate_harmonic* shall be called once the computation of *compute_hyper_ball* has finished. These methods use the formed matrix to approximate the 3 centrality measures presented in [2] in the way that is presented in the same paper.

```
0   c[−], an array of n HyperLogLog counters
1
2   function union(M: counter, N: counter)
3     foreach i < p begin
4       M[i] ← max(M[i], N[i])
5     end
6   end; // function union
7
8   foreach v ∈ n begin
9     add(c[v], v)
10  end;
11  t ← 0;
12  do begin
13    foreach v ∈ n begin
14      a ← c[v];
15      foreach v → w begin
16        a ← union(c[w], a)
17      end;
18      write ⟨v, a⟩ to disk
19      do something with a and c[v]
20    end;
21    Read the pairs ⟨v, a⟩ and update the array c[−]
22    t ← t + 1
23  until no counter changes its value.
```

The picture above is the pseudo-code of the implemented algorithm in *ComputeHyperBall*, as describe in the research paper [2].

- ***counting package:*** the package contains three python files, which implements respectively the *Flajolet-Marting* algorithm, the *HyperLogLog* algorithm and there's a *mainCounting* file to run both the algorithms and see the results of the computation.

- ***FlajoletMartin:*** the file contains the method to construct the object *FlajoletMartin* of the respective class and to run the algorithm. The implementation of the algorithms is inspired by the original paper [4]. It has the following methods:
  - *init:* the constructor instantiates the object by specifying the range of the number of elements for the multiset that it will receive as input, and two parameters l and k, which are used for computing hash functions and represent the size of the group and the number of distinct groups, respectively.
  - *estimate_ cardinality*: the function is used to estimate the approximate cardinality and leverages on the other functions of the class. It uses the k per l hash functions to improve the accuracy of the calculation. Indeed, they are split into k groups, each of size l and for each group it calculates the median of the results and then retain the mean of the k groups to obtain the final estimate.
  - *calculate_ r*: it is responsible to find the max *r(a)*, hence the maximum number of tails 0's between all the elements.
  - *lsb_ index*: it returns the index of the LSB element with 0-index positioning, from the right. It is like finding the number of tails 0's.
  - *hash_ element*: it hashes an element between 0 and $2^L - 1$ by leveraging on the hash function $h(x) = (ax + b) \% n$

- ***HyperLogLog:*** the file contains the method to construct the object *HyperLogLog* of the respective class and to run the algorithm. The implementation of the algorithms is inspired by [2] but in particular from the original paper [5]. It has the following methods.
  - *init:* the constructor instantiates the object by leveraging on the number of buckets the user would like to use. Thus, it creates attributes for storing the number of bits, number of registers, initialise the array M (register counters) to zero and contains a dictionary for the different values of alpha, as defined in the paper.
  - *add:* the method, for each element of the input multiset, hash the element by using the custom function, calculates the j index by adding one to the first b bits and w, as remaining bits. Finally, it stores in the array the maximum value between the old value in the register and the new lsb index position of w.
  - *calculate_ cardinality:* the function estimates the approximate cardinality by calculating Z, E and E^* by depending on the case. For each specific case, there are some support functions which help at handling the case. In particular, we need to compute the registers with zero elements *(calculate_ zero_ registers),* make a small and large corrections on the different value computed by using the following functions: *small_ range_ corrections, large_ range_ corrections.*
  - *union:* the function consists in obtaining the maximum for each pair of registers
  - *lsb_ index:* the same as the previous one
  - *hash_ element_ md5:* the function hashes the element in input by using the md5 standard library hash function in the range $\{0,1\}^{32}$.

- ***mainCounting:*** the files contains a main function for testing the functionalities of the two algorithms calculating distinct elements. It creates a *Multiset* object and add random elements to it. Later, it creates the object for the *FlajoletMartin* and the *HyperLogLog*. Finally, it runs the two algorithms and prints the results on the screen.

Let $h : \mathcal{D} \to \{0,1\}^{32}$ *hash data from* $\mathcal{D}$ *to binary 32–bit words.*
Let $\rho(s)$ *be the position of the leftmost 1-bit of* s*: e.g.,* $\rho(1\cdots) = 1$, $\rho(0001\cdots) = 4$, $\rho(0^K) = K+1$.
define $\alpha_{16} = 0.673$; $\alpha_{32} = 0.697$; $\alpha_{64} = 0.709$; $\alpha_m = 0.7213/(1+1.079/m)$ for $m \geq 128$;
Program HYPERLOGLOG (**input** $\mathcal{M}$: multiset of items from domain $\mathcal{D}$).
assume $m = 2^b$ with $b \in [4\,..\,16]$.
initialize a collection of m registers, $M[1], \ldots, M[m]$, to 0;

for $v \in \mathcal{M}$ **do**
  set $x := h(v)$;
  set $j = 1 + \langle x_1 x_2 \cdots x_b \rangle_2$; {*the binary address determined by the first b bits of* x}
  set $w := x_{b+1} x_{b+2} \cdots$;
  set $M[j] := \max(M[j], \rho(w))$;
compute $E := \alpha_m m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$; {*the "raw" HyperLogLog estimate*}
if $E \leq \frac{5}{2}m$ **then**
  let V be the number of registers equal to 0;
  if $V \neq 0$ **then set** $E^\star := m \log(m/V)$ **else set** $E^\star := E$; {*small range correction*}
if $E \leq \frac{1}{30} 2^{32}$ **then**
  set $E^\star := E$; {*intermediate range—no correction*}
if $E > \frac{1}{30} 2^{32}$ **then**
  set $E^\star := -2^{32} \log(1 - E/2^{32})$; {*large range correction*}
return *cardinality estimate* $E^\star$ *with typical relative error* $\pm 1.04/\sqrt{m}$.

The picture above is the pseudo-code of the implemented algorithm in *HyperLogLog*, as describe in the research paper [5].

- ***main:*** this file is the entry point of the project. In this file the graph object is constructed by *GraphReader* through the *read_graph* method call. Subsequently, the object hyperball is constructed and by calling *compute_hyper_balls* the matrix of cardinalities is calculated. In the end is possible to call the 3 centrality measure methods on the hyperball object by passing the node object that is under investigation.
-

The program has been tested by using a dataset that contains the connections of airports in the US, with almost 500 nodes and approximately 2000 edges as an undirected graph.

3. How to Run the Code

In this section it is explained how to run the code provided with this short report. The recommended environment is Ubuntu >= 18.04 LTS. It is also requested to have installed on the machine Python3 with the following dependencies: *NumPy* and *Multiset*.

The pipeline can be run by using the following command in the directory *src*.

```
$ python3 main.py
```

It is possible to use command line arguments to set specific values for the node to show and the number of buckets. The default configuration is the one below.

```
$ python3 main.py -n 10 -b 32
```

It is also possible to test the two counting algorithms by running the following command in the *counting* directory.

```
$ python3 mainCounting.py
```

4. Results

By running the program, it is possible to see the following output which shows each operation correctly executed. The first block of results is for the *Hyperball* algorithms., while the second one to the two counting algorithms with respect to the true cardinality of distinct elements.

```
$ python3 main.py
HyperBall algorithm for Mining Data Streams
Adriano Mundo & Edoardo Realini
KTH Royal Institute of Technology - 2020

Number of nodes:  10

Start the HyperBall algorithm

Reached convergence in 8 iterations
Node 0 Closeness:  0.0009199632014719411
Node 0 Lin:  290.5648574057038
Node 0 Harmonic:  332.8333333333333
Node 1 Closeness:  0.0009041591320072332
Node 1 Lin:  285.57323688969257
Node 1 Harmonic:  323.3333333333333
Node 2 Closeness:  0.0009671179883945841
Node 2 Lin:  305.458413926499
Node 2 Harmonic:  338.0
Node 3 Closeness:  0.0008841732979664014
Node 3 Lin:  279.2608311229001
Node 3 Harmonic:  310.8333333333333
Node 4 Closeness:  0.000856898029134533
Node 4 Lin:  270.64610111396746
Node 4 Harmonic:  312.4166666666667
Node 5 Closeness:  0.0009107468123861566
Node 5 Lin:  287.6539162112933
Node 5 Harmonic:  327.3333333333333
Node 6 Closeness:  0.0009416195856873823
Node 6 Lin:  297.4048964218456
Node 6 Harmonic:  331.6666666666667
Node 7 Closeness:  0.0008445945945945946
Node 7 Lin:  266.76013513513516
Node 7 Harmonic:  304.6666666666667
Node 8 Closeness:  0.0008389261744966443
Node 8 Lin:  264.9697986577181
Node 8 Harmonic:  299.9166666666667
Node 9 Closeness:  0.0009000900090009
Node 9 Lin:  284.28802880288026
Node 9 Harmonic:  320.8333333333333

End the HyperBall algorithm, time elapsed:  0.18562722206115723  sec.
```

```
$ python3 mainCounting.py
True cardinality:  1000
FlajoletMartin cardinality approx:  1024
Time elapsed (seconds):  3.949
HyperLogLog cardinality approx:  994.0
Time elapsed (seconds):  0.004
```

5. Optional Task

What are the challenges you have faced when implementing the algorithm?
The main problem we faced was discovering that the *HyperBall* paper uses *HyperLogLog*, which turns out not to be *Flajolet-Martin* algorithm. They are two different algorithms for counting distinct elements in a stream of data. Hence, it was not possible to adapt the algorithms we saw during lecture to perform the Hyperball algorithm, which works on stream graph. Thus, we had to put additional effort in the assignment for implementing both the algorithms, understand the differences and how they are related. Even it was challenging, at the end we were more than happy with the final results.

Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.
The algorithms can be parallelized without any problems, like it is possible to do in another libraries which handles scalable graph computation such as Pregel and PowerGraph. We can instantiate the vertex parallelization of the graph for every iteration t and at each step t, the results from the previous step can be available in a shared memory.

Does the algorithm work on unbounded graph streams? Explain.
If we consider unbounded graph streams like infinite data-streams, we can state that the *Flajolet-Martin* algorithm can easily handle the growth of the dataset, since it computes the cardinality for streaming data, while using *HyperBall* we have to recompute all the cardinalities from the beginning.

Does the algorithm support edge deletions? If now, what modifications would it need? Explain.
We will consider the *HyperBall* algorithm. In the case of edge deletions, it is necessary to recompute all the cardinalities from the instant t=1 without considering the removed edge. This is a must because all the cardinalities are computed as approximations, hence the algorithm need to be run again. To avoid that, a modification could be to save all the possible combinations of edges coming from a particular neighbour, but it will be expensive at memory level and not scalable. We conclude that it is not possible to allow edge deletion from the dataset by using *HyperBall*.

References

[1] Mining Data Streams lectures 5-6, ID2222, Data Mining, KTH Royal Institute of Technology

[2] P. Boldi and S. Vigna, "In-Core Computation of Geometric Centralities with HyperBall: A Hundred Billion Nodes and Beyond," 2013 IEEE 13th International Conference on Data Mining Workshops, Dallas, TX, 2013, pp. 621-628, doi: 10.1109/ICDMW.2013.10.

[3] Chapter 4 in *Mining of Massive Datasets,* by Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman, 2nd edition, Cambridge University Press, 2014

[4] Philippe Flajolet and G. Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. 31, 2 (Sept. 1985), 182–209. DOI:https://doi.org/10.1016/0022-0000(85)90041-8

[5] Flajolet, Philippe & Fusy, Eric & Gandouet, Olivier & Meunier, Frédéric. (2012). HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm.