Homework 3

Mining Data Streams

Denys Tykhoglo

Dataset

For this homework, I used the graph called Amazon (MDS) from KONECT (http://konect.uni-koblenz.de/networks/com-amazon). The nodes of the graph are integer numbers that represent Amazon products IDs and each edge between two nodes shows that the products are frequently bought together. The edges are unweighted and undirected. There is a total of 334863 nodes and 925872 edges.

The graph is represented by a text file where each line contains a single edge. Each edge is represented by a pair of node IDs that it connects.

Solution

I decided to implement the enhanced Flajolet-Martin algorithm called HyperLogLog (HLL) for counting distinct elements of a stream and the HyperBall algorithm for approximate calculating of geometric centralities. The solution consists of two parts that correspond to the assignment's tasks.

The first part reads sequentially the graph file edge by edge and counts the number of unique node IDs using the implemented HyperLogLog algorithm. For the purposes of testing the HyperLogLog implementation, sequential reading of edges from the disk is equivalent to processing the incoming edges in form of a stream. The functionality of a counter is implemented in class *HyperLogLogCounter.* Each node of the graph is added to the counter using the *add()* method, and when there are no more nodes left, the *size()* method is invoked to obtain the number of nodes. It is possible to change the approximation's precision by increasing or decreasing the number of registers used by the counter.

The second part starts from reading all nodes into the memory as a set. Then the set is processed by the method *calculateGeometricCentralities()* from class *HyperBall* with the purpose to estimate each node's cardinality metrics (that is, to determine if the node is located in the center of the graph or near its periphery). The key idea of *HyperBall* is relying on several instances of *HyperLogLogCounter* to estimate the sum of distances between a node and all other nodes. Three different metrics for centrality are estimated: closeness centrality, Lin's centrality and harmonic centrality. Eventually, the *printCentralities()* method writes the obtained estimations to a csv file.

The solution is implemented using Java SE 8.

How to run

In order to launch the solution, it is only necessary to compile the attached file *MiningDataStreams.java* and then run class *MiningDataStreams*.

The optional console parameters are the following:

1. The path to a text file that contains the graph (string). The default path is *src\main\resources\com-amazon.ungraph.txt.*
2. Parameter b for the HyperLogLog algorithm (integer number). 2^b is the number of used registers The default value is *5*.
3. The path to the output csv file where the metrics are written (string). The default path is *src\main\resources\Node centralities.csv.*

Results

Figure 1 and *Figure 2* are the screenshots of console outputs produced by the execution. *Figure 3* shows the beginning of the resulting csv file with the estimated centrality metrics for each node. Int the picture, the nodes are sorted in the descending order according to the harmonic centrality.

```
Counting the approximate number of distinct nodes at the text file using 32 registers
The approximate number of distinct nodes is 350071
Loading the nodes from the text file into main memory
Starting the HyperBall algorithm
Calculating ball sizes of radius: 1
Calculating ball sizes of radius: 2
Calculating ball sizes of radius: 3
Calculating ball sizes of radius: 4
```

Figure 1: Screenshot of the output's first part

```
Calculating ball sizes of radius: 40
Calculating ball sizes of radius: 41
Calculating ball sizes of radius: 42
Writing the centrality metrics to the output file
The centrality metrics have been written to the file successfully
```

Figure 2: Screenshot of the output's last part

Node	Closeness centrality	Lin's centrality	Harmonic centrality
222074	3.40E-07	41673.59	46887.72
98756	3.46E-07	42437.53	46735.94
199628	3.47E-07	42473.46	46731.23
537519	3.46E-07	42425.18	46426.78
35512	3.38E-07	41446.02	45913.24
239107	3.36E-07	41225.72	45789.56
145964	3.44E-07	42100.73	45789.55
54379	3.38E-07	41365.49	45515.29
341570	3.41E-07	41748.45	45508.61
317053	3.35E-07	41048.93	45495.22
13323	3.37E-07	41332.51	45267.45
231855	3.36E-07	41160.61	45229.48
239327	3.32E-07	40649.61	45151.54
25614	3.44E-07	42150.5	45054.31
154855	3.28E-07	40218.7	44947.47
89000	3.31E-07	40542.33	44927.05
449223	3.38E-07	41475.55	44831.46
110214	3.31E-07	40541.92	44811.95
284825	3.24E-07	39722.59	44782.74
53175	3.35E-07	41091.05	44711.19

Figure 3: Screenshot of the beginning of the file with estimates

Optional questions

One of the challenges that I encountered was finding a proper hash function. For the HyperLogLog algorithm, it is very important that the produced hash values are uniformly distributed, i.e. each bit of the hash should have equal probability to be 1 or 0. Initially I tried to use trivial custom hash functions that involved several multiplications, addition and modulo operations with randomly selected parameters. However, the HyperLogLog's estimation was completely incorrect almost every time as the hash functions depended greatly on the parameters' values. As the result, I opted to use secure hash function SHA-1 which is proven to have uniformly distributed value.

As for the HyperBall implementation, I changed the update file for temporary storing the updated counters to in-memory update hash map. The reason is that the graph that I am using for the assignment is relatively small, so it is feasible to store both old and new versions of the counters in the main memory. Storing the updated counters on disk has a performance drawback as the file needs to be written to and scanned for each ball size *t*, which in case of my graph is 42 times. However, the option with the update file is inevitable for larger graphs where there may not be enough main memory.

According to the authors of the original paper, the HyperBall algorithm "scales linearly with the number of available cores". They do not describe how exactly the workload is split among the cores, but I assume that one option is to create several threads (possibly, one thread per core) and assign each thread an equal number of nodes to process. The main idea is that when balls of size *t* are calculated, the nodes do not update their internal counters until all nodes complete their calculation. That means that each node's calculations are independent of all other nodes'

calculations for any given ball size *t.* But because the nodes need to update their counters to move *to t + 1,* the program must wait until all threads are ready with the balls of size *t*.

HyperLogLog algorithm can work with unbounded data streams. The value of each incoming node is added to the counter and the stream cardinality is estimated on demand and very fast. Several registers (at least 16) of one byte each can be used to give estimations for streams of a technically unlimited size. On the contrary, HyperBall cannot work with data streams as it requires the whole graph to be available from the beginning, for example at a hard drive. It includes several iterations over all nodes, and during each iteration the node uses the information from its neighbors.