Data Mining - Assignment 3

Group 18
Gustavo Teodoro D. Beck - Fernando García Sanz

November 22, 2020

1 Data

In order to test the later on explained system, several directed graph datasets obtained from the *Stanford Large Network Dataset Collection* [1] have been used. The specific datasets are:

- email-Eu-core
- CA-GrQc

2 Method

The aim of the proposed method is to approximate the centrality of the nodes that compose the graph. This is done by means of the harmonic centrality equation:

$$\sum_{y \neq x} \frac{1}{d(y,x)} = \sum_{d(y,x)<\infty, y \neq x} \frac{1}{d(y,x)} \tag{1}$$

This formula returns the harmonic centrality for a node x, being y a different node belonging to the set of all nodes that can reach x (thus, their distance is lower than infinity).

In order to compute an approximation to the real centrality, the employed algorithm consists of two sections, the *HyperLogLog counters* and the *HyperBall* algorithm.

2.1 *HyperLogLog* counters

HyperLogLog [2] is a probabilistic algorithm devoted to estimating the number of distinct elements (cardinality) of data streams. For large datasets, HyperLogLog performs (time and accuracy-wise) well since with a single pass over the data it estimates cardinality.

In essence, the algorithm reads sequentially the elements of a stream. The probabilistic characteristic of the algorithm relies on randomization, which is ensured by the use of a single fixed hash function that assimilates hashed values to infinite binary strings of $\{0,1\}^\infty$. As presented in the paper, we ought to hash on 32 bits and short bytes of 5-bit length (b), which suffice to estimate cardinalities up to 10^9.

To initialize the algorithm, it iss required to create a zero multiset $\mathcal{M}$ with size $m = 2^b$, and stream the data sequentially following these steps:

1. Hash the value in a 32-bits manner with the hash functions.
2. Get the binary address (j) determined by the first b bits of the hashed value. For the remaining bits, compute the leading zeros (w) plus one.
3. The address j corresponds to the position on the $\mathcal{M}$, which will be updated based on the maximum value between its current value and the amount of leading zeros.

Once all the elements have been hashed and allocated, the algorithm computes an indicator that returns a normalized version of the harmonic mean of the $2^{M[j]}$ performed by:

$$Z := \left(\sum_{j=1}^{m} 2^{-M^{(j)}} \right)^{-1}$$

$$E := \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-M^{(j)}}}$$

Where α_m,

$$\alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

The output E is interpreted as an estimation of the cardinality of the stream, which might not be perfect, but very accurate and fast since the data domain can increase and the computational time does not increase exponentially in comparison to the algorithm that computes the exact cardinality.

Our implementation of the algorithm can be seen in Appendix A1.

2.2 *HyperBall*

The *HyperBall* algorithm [3] uses a dynamic programming approach. It performs computations with nodes at distance t of a given node. In this case, it has been used to estimate balls of radius t around the nodes, based on the following expression:

$$\mathscr{B}_G(x, 0) = \{x\} \tag{2}$$

$$\mathscr{B}_G(x, r+1) = \bigcup_{x \to y} \mathscr{B}_G(y, r) \cup \{x\} \tag{3}$$

Thanks to this, the balls can iteratively be computed, saving the previous sets in order to be reloaded later.

It is important to highlight the need for reverting the graph since we want to know the amount of *coreachable* nodes from a particular node; this will determine its importance. Equation 1 calculates the centrality given the distance from all nodes y to a given node x. Via reverting the graph, all outgoing edges become incoming.

The HyperBall algorithm uses the previously described counters in order to perform an approximation of the set $\mathscr{B}_G(x, t)$ for each node, as keeping track of this is not feasible. This algorithm works in the following way:

1. Given an array of n HyperLogLog counters, being this the number of the elements in the stream, add to each counter a different node.
2. Initialize the radius t equal to one.
3. For each node in the stream, temporally copy its counter.
4. Compute the union of the node counter with the counters of all other nodes connected to it in a sequential manner.
5. Once all unions for all nodes have been computed, if any counter changed its value, increase the radius by one and go back to 3, otherwise, finish.

Our implementation of the algorithm can be seen in Appendix A2.

2.3 *Results*

Two datasets have been used to test our implementation. The dataset "email-EU-Core" is composed of 1005 nodes and 25571 edges, while the dataset "CA-GrQC" is composed of 5242 nodes and 14496 edges. Our assumption is that once the graph increases the HyperBall implementation should outperform - in terms of computational time - the exact brute force computation of the centrality (implementation in Appendix A3).

Dataset	HyperBall time	Brute-f. time	RMSE	Real Diameter	Appr. Diameter
email-EU-Core	24.68 s	16.79 s	17.10	7	7
CA-GrQc	36.03 s	1324.23 s	41.79	17	17

Table 1: HyperBall results comparison to brute force approach.

It can be seen that when the number of nodes is big, HyperBall performs way faster, returning a good approximation of the diameter of the graph and the centrality (RMSE between the harmonics of both methods). Nevertheless, when the graph has way fewer nodes, it takes a bit longer than the brute force approach. This proves that HyperBall is a really good approach when the amount of data received is big.

3 *Extra Points*

3.1 *What were the challenges you have faced when implementing the algorithm?*

Our biggest challenge was with regards to the HyperLogLog counter algorithm, which was poorly explained in the HyperBall paper. Therefore, we had to read the HyperLogLog paper in order to comprehend how they were hashing the values of the stream and converting its binary representation into the binary address j and the leading zeros of the remaining binary representation. In addition, in the HyperLogLog paper, it was clearer how to assign the values of the α_m and the short bytes of b length. Both variables have a significant impact on the performance of the HyperLogLog algorithm.

3.2 *Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.*

The HyperLogLog algorithm can be parallelized, but the HyperBall cannot. The addition of elements and the computation of the size in the HyperLogLog algorithm can be done in parallel, speeding up the process. Nevertheless, the HyperBall algorithm requires a sequential update, as changes in a counter value must be propagated to the next iterations over the edges. Therefore, as these operations are not independent, they cannot be parallelized.

3.3 *Does the algorithm work for unbounded graph streams? Explain.*

HyperBall algorithm, as presented in the original paper, does not work for unbounded streams since the HyperLogLog algorithm computes the counters by a single sequentially path over all of the data and stores it in the memory so the centrality can be computed dynamically. If any element is added to the graph, the counter would have to be recomputed and the HyperBall would have to be notified about restarting the centrality computation.

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

It cannot work with edge deletion. If some edges are deleted, the system would need to start over since some previously computed paths would be modified. Moreover, it does not only affect one

node, but all the paths that were computed using that edge, so the counters related to many nodes should be modified. It would be necessary to keep track of which edges were used in order to compute the updates for each node, so they can be recomputed if edges used for it are deleted. Nevertheless, this, for sure, would make the algorithm way less efficient.

4 How to Run

In order to execute the code, having the dataset inside a `data` folder, it is just needed to execute the `DataStreams.py` class via `python3 DataStreams.py` command.

The expected outputs are: times to execute the HyperBall algorithm and the Brute force centrality measurement, the RMSE between both harmonics, and the longest-shortest path found by both algorithms.

References

[1] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[2] P. Flajolet, Éric Fusy, O. Gandouet, and et al., "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," 2007.

[3] P. Boldi and S. Vigna, "In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond," *CoRR*, vol. abs/1308.2144, 2013.

Appendix

```python
class HyperLogLog:
    def __init__(self, bits, precision):
        self.bits = bits
        self.p = pow(2, self.bits)
        self.precision = precision
        self.modulus = pow(2, self.precision) - 1
        self.max_r = [0 for _ in range(self.p)]

        # Get alpha
        def integral(u):
            return pow(math.log2((2 + u) / (1 + u)), self.p)

        self.alpha_approx = pow(self.p * quad(integral, 0, np.inf)[0], -1)

    @staticmethod
    def hashsing(x):
        return hash(str(x)) & 0xFFFFFFFF  # 32-bit hash of the string value of node ID

def countLeadingZeros(self, x):
    rho = self.precision - self.bits - x.bit_length() + 1

    if rho <= 0:
        raise ValueError("Overflow")

    return rho

def rightmost_t_bits(self, number):
    mask_left = pow(2, self.precision - self.bits) - 1
    mask_right = pow(2, self.bits) - 1
    left_num = number >> self.bits & mask_left
    right_num = number & mask_right

    return left_num, right_num

def addElem(self, node):
    hashed_node = self.hashsing(node)
    remaining_bits, i = self.rightmost_t_bits(hashed_node)
    rho_plus = self.countLeadingZeros(remaining_bits)
    self.max_r[i] = max(self.max_r[i], rho_plus)

def computeEstimate(self, E):
    E_star = 0
    if E <= (5 / 2 * self.p):
        V = len(np.where(np.array(self.max_r) == 0)[0])
        if V != 0:
            E_star = self.p * math.log2(self.p / V)
        else:
            E_star = E
    elif E <= (1 / 30 * pow(2, self.precision)):
        E_star = E
    elif E > (1 / 30 * pow(2, self.precision)):
        E_star = -pow(2, self.precision) * math.log2(1 - E / pow(2, self.precision))

    return E_star

def computeSize(self):
    z = sum([pow(2, -max_r) for max_r in self.max_r])
    E = self.alpha_approx * pow(self.p, 2) * (1 / z)

    return self.computeEstimate(E)
```

Figure A1: HyperLogLog implementation.

```python
def hyperball(graph, bits, precision):
    print("Graph reverted")
    edges = [(v, w) for (v, w) in tqdm(graph.edges)]
    cnt = {}
    harmonic = {}
    for node in tqdm(graph.nodes):
        cnt[node] = HyperLogLog(bits=bits, precision=precision)
        cnt[node].addElem(node)
        harmonic[node] = 0
    radius = 1  # Threshold radius
    print("Starting value comparisons...")
    while True:  # While value changes
        change = False  # False --> no value changed
        cnt_old = copy.deepcopy(cnt)
        for (v, w) in edges:
            a = cnt[v]
            new_cnt = union(a, cnt_old[w])
            if new_cnt.computeSize() != a.computeSize():
                change = True
            cnt[v] = new_cnt
        harmonic = estimate_centrality(harmonic, cnt, cnt_old, radius)
        print("Radius", radius)
        if not change:
            break
        radius += 1
    print("Value comparisons done")

    return cnt, harmonic, radius

def union(cnt_1, cnt_2):
    new_cnt_1 = copy.deepcopy(cnt_1)
    for i in range(len(cnt_1.max_r)):
        new_cnt_1.max_r[i] = max(cnt_1.max_r[i], cnt_2.max_r[i])

    return new_cnt_1

def compute_diff(cnt, cnt_old, harmonic, radius, x):
    diff = cnt.computeSize() - cnt_old.computeSize()
    harmonic += diff / radius

    return {x: harmonic}

def estimate_centrality(harmonic, cnt, cnt_old, radius):
    pool = multiprocessing.Pool(multiprocessing.cpu_count())
    result = pool.starmap(compute_diff, [(cnt[x], cnt_old[x], harmonic[x], radius, x) for x in cnt.keys()])
    pool.close()
    harmonic = {k: v for d in result for k, v in d.items()}

    return harmonic
```

Figure A2: HyberBall implementation.

```python
def computeHarmonic(graph):
    harmonic = {node: 0 for node in graph.nodes}
    short = {node: 0 for node in graph.nodes}
    for x in tqdm(nx.nodes(graph)):
        for y in nx.nodes(graph):
            if x != y:
                try:
                    shortest_path = nx.shortest_path(graph, y, x)
                    shortest_path = len(shortest_path) - 1
                    short[x] = max(short[x], shortest_path)
                    harmonic[x] += 1 / shortest_path
                except:
                    pass

    return harmonic, short


def rmse(dict1, dict2):
    error = 0
    for k, v in dict1.items():
        error += pow(v - dict2[k], 2)

    return pow((error / len(dict1)), 0.5)
```

Figure A3: Exact centrality implementation.