Project HyperBall

Jakub Stachurski

January 26, 2024

Contents

1 Introduction

The world is a massive place with billions of people. And yet, it seems that you can reach any person via just a handful of people. Those small world effects were proven to affect real interpersonal networks trough experiments [Milgram, 1967] and through analysis of existing data [Backstrom et al., 2012]. In this project, we will see how the latter is done and implement some of the algorithms that make it possible to prove that a big world can be small in some sense.

2 Research: Calculating distances

2.1 Problems with big data

The research about those effects from Facebook [Backstrom et al., 2012] analyses the connections between ≈ 721 million Facebook users, which have ≈ 69 billion links between them. The sheer size of this dataset would dwarf any system at the time. The dataset used by [Backstrom et al., 2012] was around 345GB, this is more than a consumer device can handle at once. The team needed to find a way to make the dataset smaller and process it in a way which streams the data from disk instead of needing it to be all in RAM.

2.2 Solutions

The first that was done to the Facebook dataset is improving the compression by improving locality of the graph *labelled layer propagation* [Boldi et al., 2011]. This reduced the size of the dataset but also took 10 days to compute [Backstrom et al., 2012]. Because of the long compute time and additional complexity, this project will not include LLP in any shape or form.

The second solution that the Facebook team came up with is using Probabilistic Counters in the form of the HyperLogLog [Flajolet et al., 2007] algorithm to calculate the amount of unique nodes in a set, without the need to keep the whole graph in RAM.

2.3 HyperLogLog

The HyperLogLog algorithm is an extension of the LogLog algorithm, with the biggest distinction being that HyperLogLog uses the harmonic mean instead of a geometric mean for the final calculation. The LogLog and the HyperLogLog algorithm exploit Probabilistic Counting based on the amount of 0's before the first 1 in the hash ($p : \{0,1\}_n \to Z_{>0}$) to figure out the cardinality of the set they get passed in. [Flajolet et al., 2007]

The HyperLogLog counters work using a hash function of the inputs that returns a string of bits: $h : D \to \{0,1\}_n$ where n is any specified amount of bits. In most cases, it's either 32 or 64 This hash is split into two parts:

1. h_b: b leftmost bits, used as the index into the registers

2. h^b: b rightmost bits, used for the value to put into the register

Because we got b bits as the index into the registers, M_i, we need to have precisely 2^b registers to accommodate that. The registers make the counting more efficient, because instead of just testing the whole hash, we split the set and averaging the multiple tests and benefit from averaging effects. [Flajolet et al., 2007] In the case of taking 11 bits from the hash, this means $m = 2^1 1 = 2048$ registers. Each register needs to have $log_2 log_2 N + 1$ bits to accommodate a cardinality of $\leq N$ [Flajolet et al., 2007], so 5 bits is more than enough for graphs with more than 10^9 nodes. So if we take single byte registers for ease of use, we come out to a memory use of 2 KiB for the registers, which is extremely efficient. Especially to storing hashes in a HashSet.

We can add a multiset $\mathscr{M}$ the counter by adding each element to the counter as follows:

Algorithm 1 Adding an item to a HyperLogLog counter

function ADD(M, x)
  $j \leftarrow h_b(x)$
  $M[j] \leftarrow max\{M[j], p(h^b(x)\}$
end function

This process can be paused and resumed by saving the registers on disk, since there is no information left about each iteration, and only the maximal p for each register is used by the algorithm anyway. [Flajolet et al., 2007]

After going through the data, we can take the harmonic mean of the registers to get the raw estimate:

$$E := \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-M[j]}} \tag{1}$$

α_m is a constant calculated from the amount of registers we have, its formula can be found in equation 4 of [Flajolet et al., 2007]. For 2048 registers the value is estimated to be $\alpha_{2048} = 0.72$

If the resulting E is between $\frac{5}{2}m$ and $\frac{1}{30}2^32$ we got the estimate, otherwise, we need to apply corrections [Flajolet et al., 2007]:

Algorithm 2 Corrections for the raw estimate of cardinality

function COUNT(C)
  let E be the result of equation (1) on counter C
  if $E \leq \frac{5}{2}m$ **then**
    let V be the amount of registers with a 0
    if $V > 0$ **then**
      $E^* \leftarrow mlog(m/V)$
    else
      $E^* \leftarrow E$
    end if
  else if $E >= -2^{32}$ **then**
    $E^* \leftarrow -2^{32}log(1-E/2^{32})$
  else
    $E^* \leftarrow E$
  end if
  return E^*
end function

This will return us the estimate of the cardinality of the multiset, which is useful for getting the unique amount of nodes in a given area.

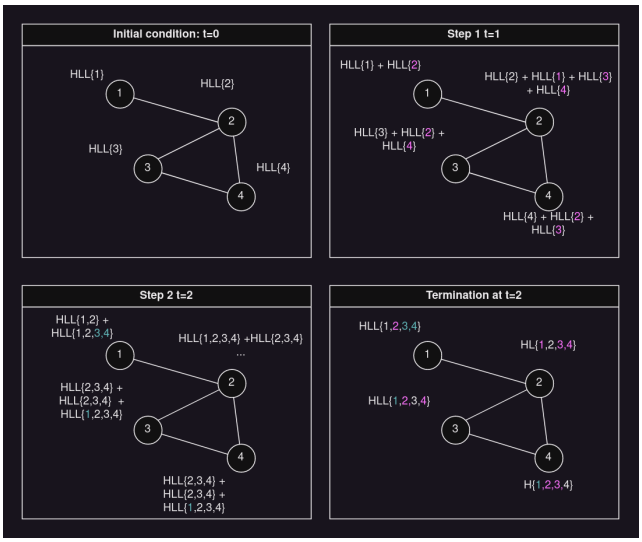

Figure 1: Step by step execution of the HyperBall algorithm

2.4 HyperBall

The HyperBall algorithm uses HyperLogLog counters to count the size of a ball of nodes $\mathscr{B}(n,r)$, where n is the center node and r is the radius. In common terms, a ball of nodes around a node n is a grouping of all the nodes that have the distance less or equal than r [Boldi and Vigna, 2013]. It does that by initializing the counters with the value of the node assigned to it. And then merging each counter with the counters of the nodes neighboring to the node assigned to that counter. This is done using the Union function. [Boldi and Vigna, 2013] Using this method, we can get the size of the balls of nodes around each node with the radius being equal to the iteration count t, starting with one. And by subtracting the size of the previous iteration, we can get the amount of nodes with distance being equal to t from each node. By aggregating this data for all nodes on each iteration, we get the frequency distribution.

Algorithm 3 HyperBall algorithm to calculate the frequency distribution

Let $c[n]$ be HyperLogLog counters for each node
Let $D[n]$ be a copy of c[n] on disk
Let $E_{-1}[n]$ be the estimate ball size of the previous iteration
Let $F[d]$ be an (dynamic) array to store the frequencies of pairs of distance d
function UNION(X,Y)
    for each $i < m$ **do**
        $X[i] \leftarrow max\{X[i], Y[i]\}$
    end for
end function
for $n \in G$ **do**
    ADD(c[n], n)
end for
$t \leftarrow 1$
repeat
    for $n \in G$ **do**
        $a \leftarrow c[n]$ (copy)
        for $n \rightarrow w$ **do**
            $a \leftarrow$ UNION(a, c[w])
        end for
        $E \leftarrow$ COUNT(a)
        $F[t] \leftarrow F[t] + E - E_{-1}[n]$
        $d[n] \leftarrow a$
        $E_{-1}[n] \leftarrow E$
    end for
    $c \leftarrow d$
    $t \leftarrow t + 1$
until no counter changes

3 Implementation

3.1 Source code

After researching the algorithm. A Rust version of the algorithm was attempted, but it was scrapped due to the limitations of importing and generating graphs. This version was replaced with a Python version. The source code is available in the following repo:

`https://github.com/Wilkuu-2/hyperball_py`

This version provides a parallelized version of the HyperBall algorithm and Breadth First Search, along with `benchmark.py` and `graph.py` utilities which relate to creating benchmark graphs.

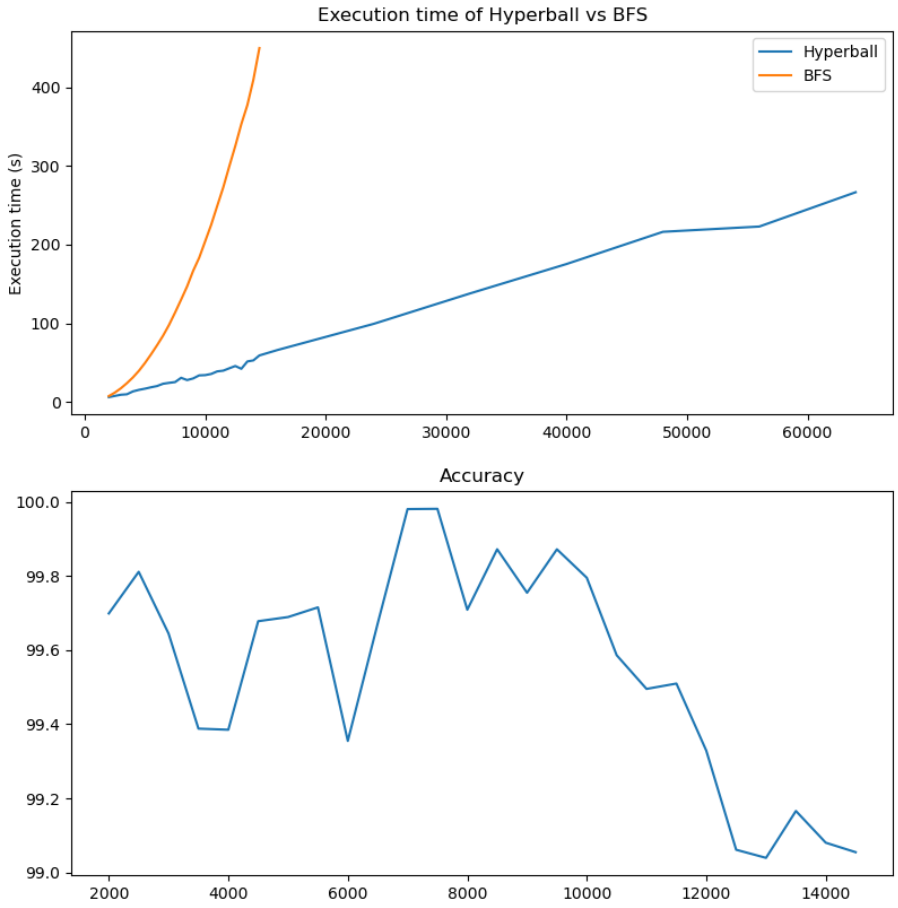

Figure 2: Benchmarks results with parallel algorithms, 8 bit precision.
Note that the time is the time it takes to compute the distribution 4 times

3.2 Results

The results of letting the two algorithms run their course on a Barabási–Albert graph can be seen here. For each amount of nodes, both algorithms run 4 times. The BFS algorithm runtime seems to increase in a quadratic manner, while Hyperball seems to be almost linear. This is expected since the algorithm only visits each node along with its neighbors once per iteration for as many iterations it takes each counter to stabilize, which is equal to the diameter of the graph. So at worst case HyperBall will perform as bad if not worse than BFS. But the only graph this occurs is a linear graph, which are uncommon in practice.

Due to the shortage in manpower, I was unable to analyze real world graphs. But in the Barabási–Albert graph that had been generated, all the graphs had the average distance of $3 \leq \bar{d} \leq 4$

4 Conclusion

Despite not getting a good measure of small world effects, we got a quick and (just a little bit) dirty way of getting distance distributions in huge graphs. HyperBall along with HyperLogLog scales way better than common algorithm, BFS, but for future work it might be good to compare other more optimized algorithms and compare them to HyperBall, along with using real data.

References

[Backstrom et al., 2012] Backstrom, L., Boldi, P., Rosa, M., Ugander, J., and Vigna, S. (2012). Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, page 33–42, New York, NY, USA. Association for Computing Machinery.

[Boldi et al., 2011] Boldi, P., Rosa, M., Santini, M., and Vigna, S. (2011). Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, page 587–596, New York, NY, USA. Association for Computing Machinery.

[Boldi and Vigna, 2013] Boldi, P. and Vigna, S. (2013). In-core computation of geometric centralities with hyperball: A hundred billion nodes and beyond. In *2013 IEEE 13th International Conference on Data Mining Workshops*, pages 621–628.

[Flajolet et al., 2007] Flajolet, P., Fusy, E., Gandouet, O., and Meunier, F. (2007). Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics amp; Theoretical Computer Science*, DMTCS Proceedings vol. AH,...(Proceedings).

[Milgram, 1967] Milgram, S. (1967). The small-world problem. *PsycEXTRA Dataset*.