HyperBall

A small program that esimates the centrality of nodes in a graph

Solution

The solution is split into its most relevant parts

Flajolet-Martin

Flajolet-Martin counters are used to estimate the number distinct items when it is not possible to count all items. This application uses Flajolet counters to estimate the number of nodes that can co-reach a certain node in the graph.

A counter is updated by the following procedure:
An item (node) is hashed into a hash-value, this application uses Jenkins hash function on the node id. This hash value is then inspected and split into two parts. One part is used for deciding which *register* to update and the other part is used to estimate the number of distinct items using trailing (or leading) zeros as a measure.

Here is an example using 16 registers (4 register bits):

```
# Hash value 68 represented as a bit string
000000000100 0101
```

The rightmost 4 bits are converted to a value and used to index a register, 5 in this case.
The remaining leftmost bits are used to count trailing zeros, 2 in this case, which is stored in to register nr 5 increased by 1.

Using more *registers* will lead to a more accurate estimatation at the cost of memory usage. Each register is initialized to `-inf` which indicates that a counter is *incomplete* or haven't seen enough distinct items.
The size of the counter can be estimated as the **number of register over the sum of 2 to the power each register value negated** multiplied by some adjusting constants. This sum becomes 0 if any register value is `-inf`.

HyperBall algorithm

The HyperBall algorithm estimates centrality by counting the distances from all nodes to a certain node. This can be done with Flajolet counters by creating a *hyperball* of increasing size (n-hop neighbors) and merging the current nodes counter with all its neighbors.

Running the hyperball algorithm with size 1 on all nodes will estimate how many nodes each node can reach with one hop. Run the exact same code again and each counter now represents how many nodes each node can reach with 2 hops. By iterating the algorithm and increasing the hypothetical ball size the program will eventually converge and the counters will estimate how many nodes are connected to each node.

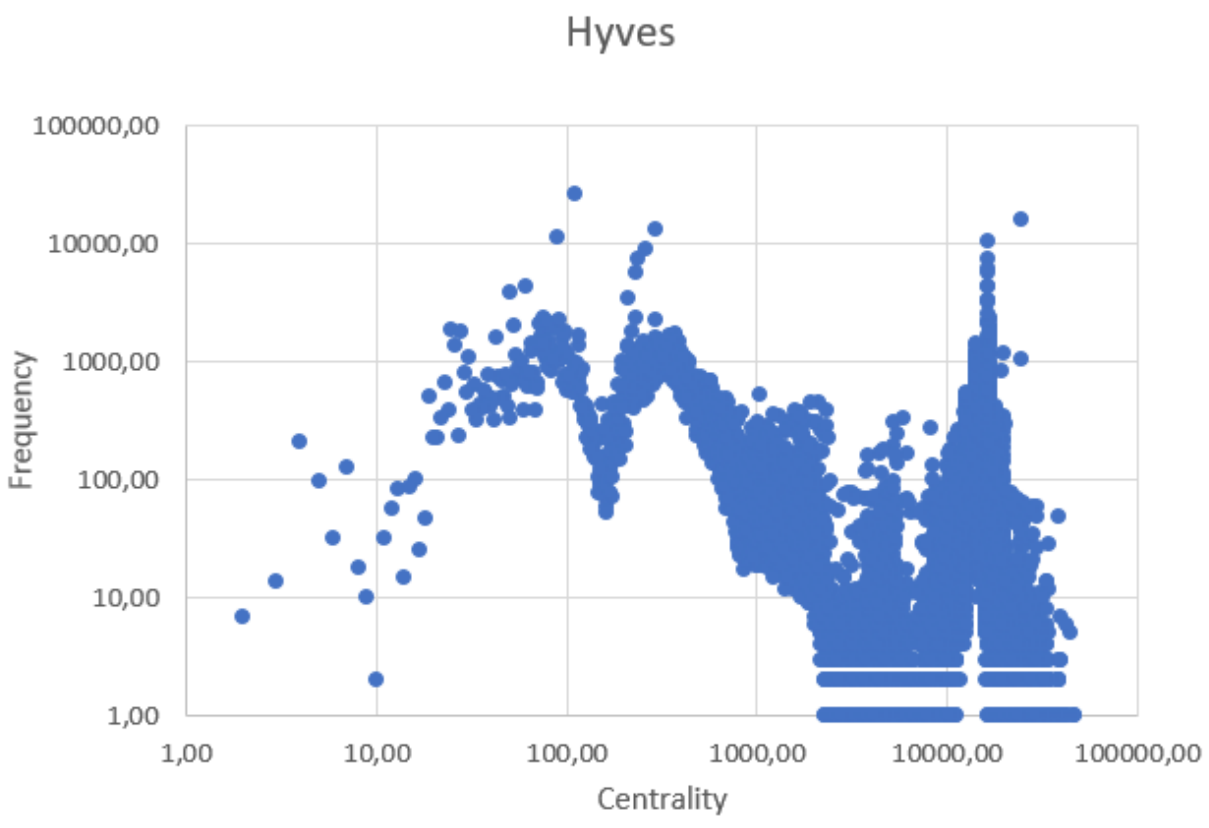
By looking at the difference between e.g. iteration 1 and 2 one can see how many nodes can reach a node with exactly 2 hops. Multiply that number with 2 (the ball size) and that's the sum of distances to a node with two hops. By repeating this it's clear that this values add up to the sum of distances to a node from every other node.

A central node will have a low sum of distances since all other nodes are considered to be *close* to that node. A counter with `-inf` in one of its registers is considered as an outlier and will have a large value according to the size function above.

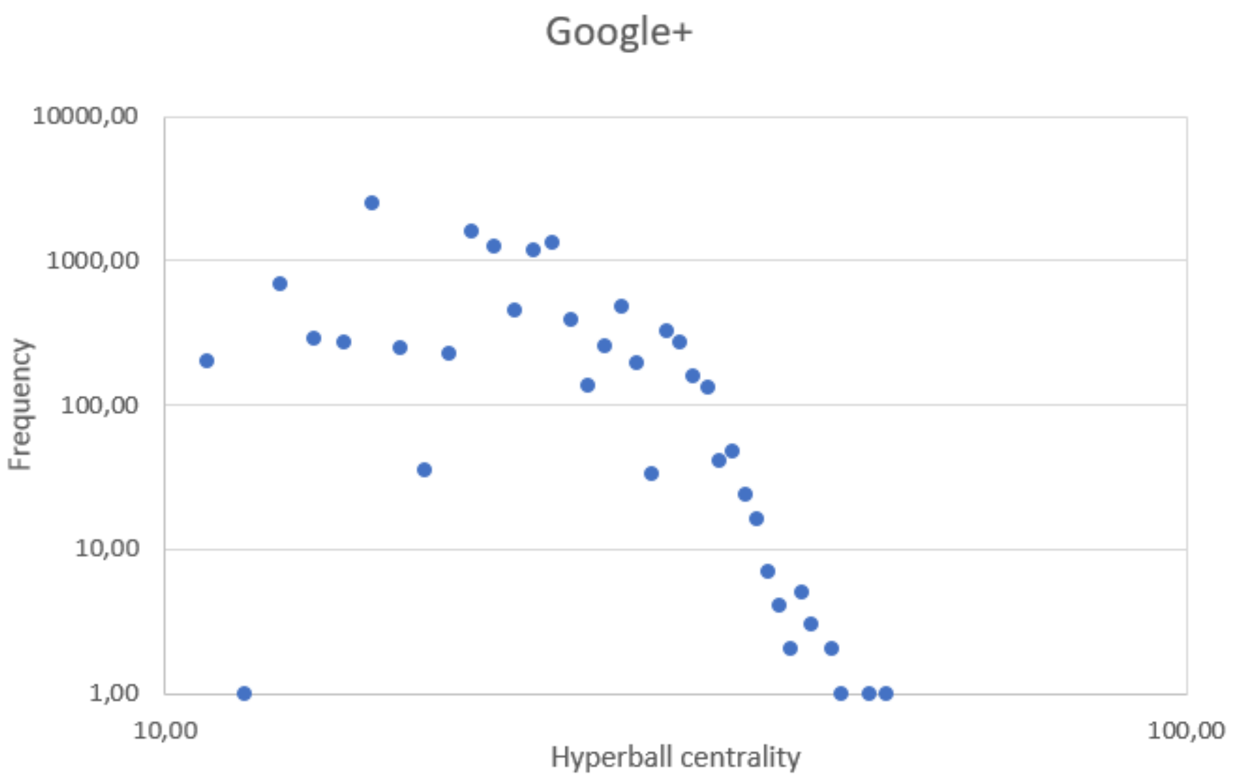
Results

To test the program, two different datasets were used, Hyves social graph and Googe+ social graph. Both centralities tend to follow loglog distribution, which was expected. The x-axis shows the centrality, higher being more central and the y-axis shows the frequency of each centrality.

Hyves



Google+



Questions

1. What were the challenges you faced when implementing the algorithm?

The biggest challenge was understanding the paper. In some places it was unnecessarily theoretical, and in many places it was poorly written with typos in formulas and pseudo code.

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

Yes, it can. A pool of threads can go through every edge in the graph one at a time and only lock the counter of a single node when calculating and writing the union of two counters. Because of the large amount of nodes, the probability that two threads have to wait for each other is very small. If the edges are sorted one could create a smarter work distribution that assigns nodes to workers instead of edges.

3. Does the algorithm work for unbounded graph streams? Explain.

No, because the algorithm runs in iterations over all edges, until the graph converges. If there is a continuous stream of edges, it will never converge.

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

No, not out of the box. Deleting an edge decreases the centrality of nodes which is not taken into account in the algorithm.
In order to support edge deletions the program must save the counters for each iteration so that a more correct centrality can be recalculated for the nodes that were connected by the deleted edge.

How to run

Prerequisites

The program is written in ruby, but uses jruby to fully utilize threading to increase speed. Jruby is highly recommended but ruby can of course be used.

Parameters

The only argument passable parameter is the file of edges in tsv format:

- `edges:` (mandatory) path to a file of edge-pairs in tsv format.

Examples

Using jruby

note: when using jruby you are required to set the jvm heap size to an appropriate value depending on amount of nodes

```
jruby -J-Xmx1024m lib/main.rb data/graph.tsv
```

Using ruby

note: if rvm (or rbenv etc) is installed, this will run as jruby because of .ruby-version

```
ruby lib/main.rb data/graph.tsv
```

The program will output the frequency of each detected centrality.

Estimated run properties: For 100k nodes and 13M edges the application used ~4GB ram and took ~10 minutes to run.