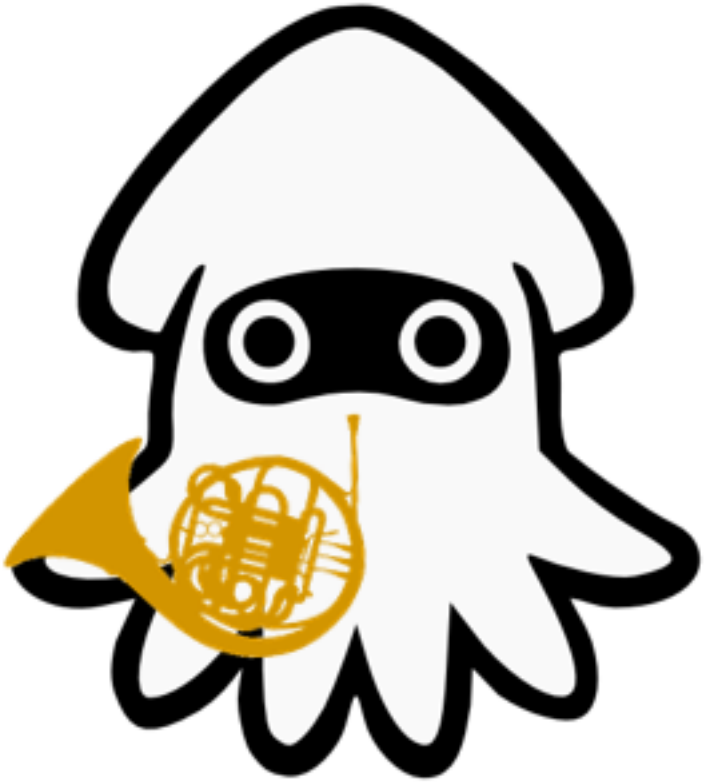
CS/EE 144: Rankmaniac 2013

Engaging a Squid's Horn

David Ding, Angela Gong, Mike Qian, and Kalpana Suraesh
California Institute of Technology
{*ding,angela,mqian,ksuraesh*}*@caltech.edu*

February 17, 2013



Contents

1 Introduction

1.1 Motivation

Our project was to use Amazon's Elastic MapReduce to quickly compute PageRanks of a large number of nodes. This was to be done in a span of 10 days with limited testing on Amazon, but unlimited local testing. The goal was to get an algorithm that would correctly rank the top 10 nodes of a graph in a short amount of time, beating the undergraduate and graduate teaching assistant teams in the CS/EE 144 class.

1.2 Team

1.2.1 Team Name

Our team name is strange, but that's because "Engaging a Squid's Horn" is an anagram of our last names: Ding, Gong, Qian, and Suraesh!

1.2.2 Team Member Roles

We divided the work as follows. All of the team members helped in coming up with and debugging various PageRank algorithms, as well as documentation.

- David Ding
  - Generated random graphs for testing purposes.
  - Took data sets online and parsed them into the proper format.
- Angela Gong
  - Created a script to automate uploading and submitting files to EC2.
  - Analyzed and plotted rank data to determine better stopping criteria.
- Mike Qian
  - Wrote the initial implementation of the PageRank algorithm.
  - Improved code quality and efficiency.
- Kalpana Suraesh
  - Optimized the implementation of PageRank to stop when the list of top N nodes stops changing.
  - Improved stopping criterion code.

2 Approach and Optimizations

2.1 Introduction

Instead of using matrix multiplication, which would be costly and take forever with a very large data set, we decided to use the iterative approach of computing PageRanks.

2.2 PageRank Algorithm

2.2.1 The Algorithm

The algorithm is split into two parts. The first part is a (possibly) multi-task PageRank algorithm which processes input from stdin and outputs it into a format that is better for PageRank calculations. The second part is a processing algorithm, which runs as a single task and does the final output (if done), or formats the output for another run.

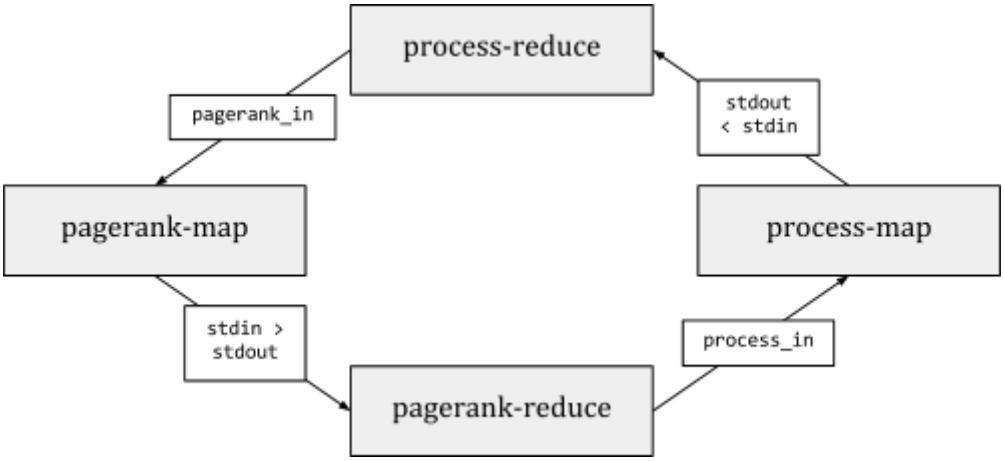


Figure 1: *Steps in an iteration of the PageRank MapReduce algorithm.*

The algorithm runs in many iterations; each iteration goes through the data once, and if more processing is required, the algorithm goes through another iteration. Details about the steps are below:

- `pagerank-map`: We receive the results of the previous iteration as inputs for the next iteration. Our results are formatted to include the relative ranks of the nodes. The very first iteration is detected by the mapper to be the first, at which point ranks are initialized to -1. Each received tuple is then in the format `(node_id, pagerank, prev_relrank, outdegree, dests)`, where pagerank is the current PageRank value, `prev_relrank` is the relative rank of the node in the last iteration, `outdegree` is its outdegree, and `dests` contains all the adjacent nodes. The received tuple is re-emitted for the reducer, and for each node $d \in$ `dests`, a tuple of the form `(d, added_rank)` is emitted, where `added_rank` is `pagerank` divided by `outdegree`. This is the first part of the iterative PageRank algorithm.

- `pagerank-reduce`: The input from the mapper has format `(node_id, values)`, where values can be in two formats: either some float f_i that represents an amount to be added to the current PageRank value, or a tuple of the form `(relative_rank, pagerank, outdegree, dests)`, defined as above. The new pagerank for each node is then calculated by adding the sum of the flows f_i, scaled by a factor of α,

and $(1-\alpha)$, as per the PageRank algorithm. For each input, the reducer finally outputs a tuple of the form `(node_id, prev_relrank, new_pagerank, outdegree, dests)`.

- `process-map`: Nothing is done here, because we cannot guarantee that only one mapper will be called, and we need to have knowledge about the entire data set to make a decision at this stage.

- `process-reduce`: This is the main optimization logic. After several runs, we determined that the best stopping criterion is to stop after the set of nodes with the top 10 ranks stays the same for two successive iterations. We implement this by enforcing a fixed-size min-heap on top of Pythons `heapq`. The max size of the heap is 10; once the heap is full, for each node that is read in, we compare the new nodes PageRank value with the PageRank value at the top of the min-heap. If the new node has a higher page rank, then we remove a node from the min-heap, and add the new node to the heap. Once all nodes are received, `process-reduce` checks if any of the ranks of nodes in the top 10 have changed; if they have not, we are done and output the final results. If they were any changes, then we must run another iteration. In this case, we emit the top 10 nodes with ranks values of 0-9, and all the other nodes with a rank value of -1.

2.2.2 Changing the Damping Factor α

After reading some papers, we realized that changing α might cause the results to converge faster, and still give results similar to that of the standard value $\alpha = 0.85$. We looked at the Caltech web graph to see what would happen if we changed α. Our runs for $\alpha > 0.85$ included our early termination algorithm (see section 2.3.3), which is why there are errors for some values of α for the top 10 nodes.

α	Iterations	# Wrong
0.85	29	0
0.86	15	1
0.87	15	0
0.88	13	0
0.89	12	0
0.90	11	0
0.91	11	0
0.92	10	0
0.93	15	7
0.94	13	7
0.95	15	8

Table 1: *Different α's and resulting number of iterations and errors.*

We noticed that between $\alpha = 0.87$ and $\alpha = 0.92$, there were no errors in the top 10 nodes, and it took about the same or fewer iterations than $\alpha = 0.85$. For run 4, we decided to use $\alpha = 0.89$, somewhere in the middle.

However, as can be seen by the change between run 3 and run 4, $\alpha \neq 0.85$ actually made the algorithm converge more slowly, and we had more errors! We concluded that increasing α does not actually work well for our algorithm.

2.3 Stopping Criterion

2.3.1 Introduction

We require a stopping criterion to ensure that our PageRank algorithm does not run forever. The assignment imposed a limit of 50 iterations, but in order for the algorithm to run faster, we want it to converge sooner to the correct result.

At first, our algorithm stopped once the residuals (the differences in ranks for each node between the previous iteration and current iteration) were all below a certain threshold. Once this worked, we refined the stopping criterion such that it would terminate the algorithm early, sacrificing some accuracy. We chose to sacrifice just the right amount such that the penalties that resulted were less than the time saved by terminating early.

2.3.2 Residuals

We started with finding residuals of the ranks. At each iteration, we stored both the new and old absolute PageRanks, and if the sum of the residuals was less than some ε (in our case, 0.01), we consider the PageRanks to be "stable" and terminate the algorithm. If we define $\mathbb{P}_{\text{old}}$ to be the old PageRank and $\mathbb{P}_{\text{new}}$ as the new PageRank, then our equation is as follows:

$$\text{residuals} = \sum_{i=1}^{N} (\mathbb{P}_{\text{old}}(i) - \mathbb{P}_{\text{new}}(i)) \leq \varepsilon$$

On both the Caltech web graph and the $G(n, p)$ graph, we noticed that setting $\varepsilon = 0.01$ seemed reasonable, and the results converged within 30 iterations. Setting ε any smaller did not make much sense because the improvement tapers off as the number of iterations increases.

2.3.3 Stability of Relative Ranks

While the residuals allowed us to terminate with good rankings, the algorithm ran for too many iterations. We printed the ranks of each node at each iteration, and noticed that the top 15 nodes' ranks became stable after a certain point, as shown in Figure 3 (which only graphs the top 10 nodes for a clearer view).

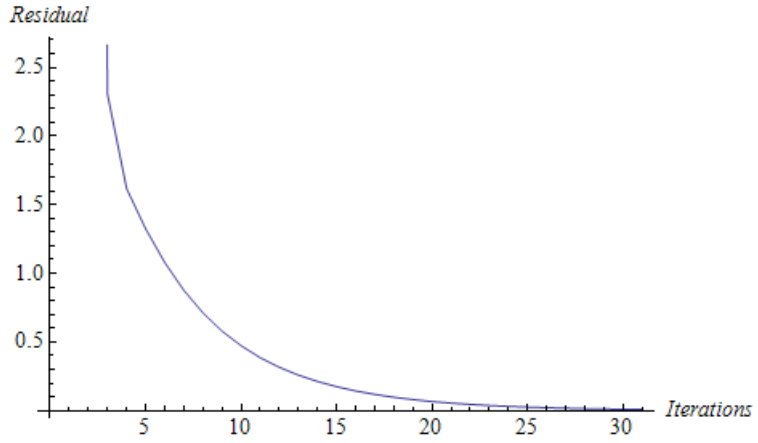


Figure 2: *Residuals of the PageRanks for nodes in the Caltech web graph.*

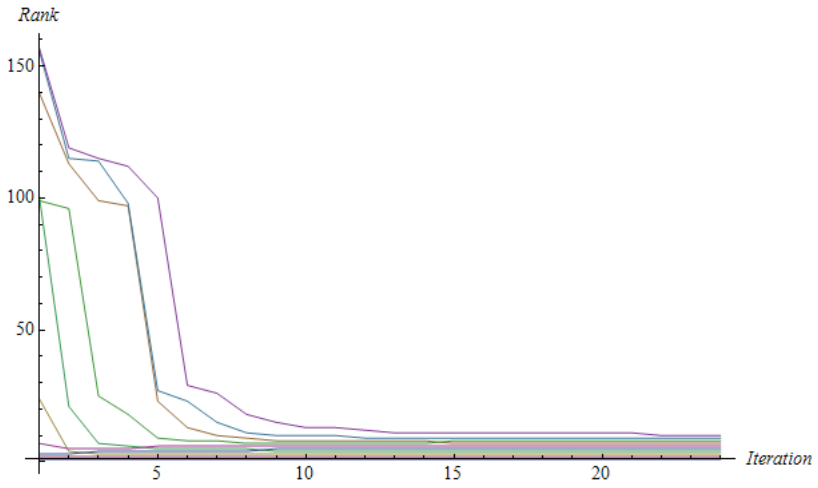


Figure 3: *Ranks of the eventual top 10 nodes for the Caltech web graph.*

Taking advantage of this, we rewrote the stopping criterion such that the algorithm would stop after the top 15 nodes remained the same between two successive iterations. This is not perfect, though, as the top 15 may remain constant for some number of iterations, then change later on, as can be seen in Figure 4. However, since we were optimizing for speed, this was acceptable, as we would swap one or two places at most, sacrificing a smaller penalty for a bigger increase in speed.

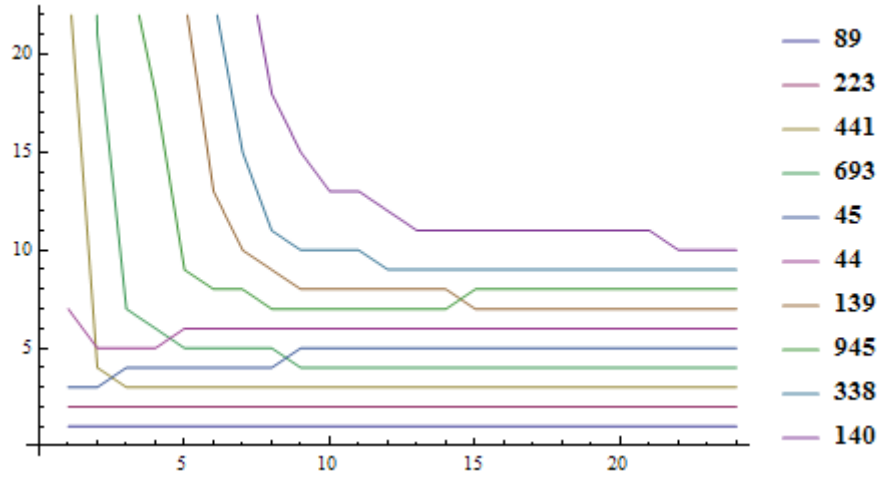


Figure 4: *Zoomed-in view of the ranks of the top 10 nodes in the Caltech web graph. The labels are the node IDs.*

2.4 Optimizing for Graph Structure

2.4.1 Sparsity

The graphs in the test data (as well as the grading data) were sparse graphs. That is, there were many nodes that had very few to no outlinks or inlinks. This means that we could have pruned the input graph to remove such nodes, since they would not contribute to the PageRank calculations much. However, we decided that the initial overhead of pruning of the graph would be much greater than the benefits from pruning.

2.4.2 Heavy-tails

Since the Caltech web graph is heavy-tailed (and likely at least one of the testing data sets), we implemented our stopping criterion to take advantage of this. If a graph is heavy-tailed, then the top few nodes will have much larger PageRanks than the rest of the nodes. Therefore, our stopping criterion of stability (section 2.3.3) works because once the top few ranks are stable and do not change, we do not expect the rank of some lower-ranked node to suddenly jump up. Assuming this, we were able to make our algorithm relatively fast.

2.4.3 Clustering

Finding the cluster coefficients of a node might help us find the PageRank of the node faster, because a node with higher clustering coefficient likely has many inlinks from which to receive PageRanks. However, we figured that calculating the clustering coefficient for each node would be costly and not worth the possible improvement.

2.5 Optimizing on EC2

There were some settings that could change the speed of our algorithm, namely, setting the number of map and reduce tasks. Amazon recommends a ratio of 2:1 map versus reduce tasks. Since we are running 10 instances in the grading set, we chose 20:10 as our ratio of mappers to reducers.

When testing with the Caltech web graph, we noticed that it took longer running 20:10 (20 mappers, 10 reducers) than it did to run 1:1. This conclusion may not be accurate because throttling occurred and our time measurements may have been wrong. However, we decided this was because of the overhead required to allocate 20:10 tasks, as well as the fact that we were running one instance of MapReduce when testing.

3 Runs

1.

02/10/13 FAILED	*Stopping criterion*: Residuals, with $\varepsilon < 0.01$ *Tasks*: 4:4 PageRank, 1:1 process

Our first run was unsuccessful. We incorrectly assumed the format of sorting on Amazon EC2. We knew Amazon sorted keys only, and nothing within the key. However, when testing locally, our inputs were keys separated by tabs, so our sorting was a little off. As a result, our algorithm terminated prematurely and errored out.

We also incorrectly assumed that specifying $1:1$ for the process MapReduce would ensure that Amazon only allocated 1 map and 1 reduce task for the process step. This was not the case, and when we ran with $4:4$ for the PageRank step, Amazon insisted on having 4 process map tasks. At this point, we realized that Amazon recommends a ratio of $2:1$ map to reduce tasks for the `m1.small` instance on EC2, so we moved all the code from `process-map.py` to `process-reduce.py` such that the process reduce step would definitely happen in one task.

2.

02/11/13 1:55:43 + 0:00:00 penalty = 1:55:43	*Stopping criterion*: Residuals, with $\varepsilon < 0.01$ *Tasks*: 20:10 PageRank, 1:1 process *Changes*: No assumption of one process map task.

After correcting our code for the incorrect assumption that the process MapReduce only runs a single map task, everything worked! However, our code ran for much longer than we expected. The undergraduate TA's code at this point took 1 hour, and ours was twice as long. We figured that the stopping criterion needed improvement so the code would terminate earlier.

3.

02/12/13 0:34:51 + 0:01:00 penalty = 0:35:51	*Stopping criterion*: Stability of top 15 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: No assumption of one process map task.

Since we already had an algorithm that gave correct code, our next task was to optimize it. We looked at how the top 15 ranks changed on the Caltech web graph, and decided to make our stopping criterion based on the changes of the top 15 ranks. If they did not change for two rounds, then we terminate the algorithm. Although this may not necessarily return the right result, it would be fast, and the number of incorrect nodes would be small so the penalties would be small as well. As we can see, this was true, as our algorithm took $\frac{1}{4}$ of the time and there was only one node (probably the 10th place) that was wrong!

4.

02/13/13 0:39:14 + 0:10:00 penalty = 0:49:14	*Stopping criterion*: Stability of top 15 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: $\alpha = 0.89$.

Oops! It turns out that having $\alpha = 0.89$ does not work for the test data set. We decided to go back to having $\alpha = 0.85$.

5.

02/13/13 0:33:14 + 0:01:00 penalty = 0:34:14	*Stopping criterion*: Stability of top 12 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: $\alpha = 0.85$; compare top 12 ranks

We figured that comparing only the top 12 ranks for stability instead of the top 15 would be faster. In fact, it was! However, it was only a minute faster, so we are not sure if it is due to the algorithm or Amazon EC2 having a lower load at that point.

6.

02/14/13 0:33:32 + 0:01:00 penalty = 0:34:32	*Stopping criterion*: Stability of top 11 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: Compare top 11 ranks

In order to cut more time, we decided to make the stopping criterion even stricter. This means that we would only compare the top 11 nodes for stability, and stop when they remained the same for two rounds. However, it did not result in any improvement in time.

7.

02/14/13 0:31:19 + 0:01:00 penalty = 0:32:19	*Stopping criterion*: Stability of top 10 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: Compare top 10 ranks

It seemed that the changes to the stopping criterion were not getting us anywhere. We decided this time to optimize the code itself. We decided that instead of outputting every node we see in `pagerank-map`, we keep a dictionary of these nodes so there are fewer things to output. When testing locally with the Caltech web

graph, the run time went from around 20 seconds to 2 seconds. However, it did not seem to make much of a difference on Amazon.

8.

02/14/13 0:39:22 + 0:00:00 penalty = 0:39:22	*Stopping criterion*: Stability of top 10 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: $\alpha = 0.86$

We decided to experiment with α again. It seems that α reduced our penalty, but increased the runtime again. It seemed that no matter how we tinkered with α or our stopping criterion, we could not get a runtime under 30 minutes!

9.

02/15/13 0:29:31 + 0:01:00 penalty = 0:30:31	*Stopping criterion*: Stability of top 10 pages *Tasks*: 20:10 PageRank, 1:1 process *Changes*: $\alpha = 0.85$; more efficient heap queue

Instead of trying to improve the stopping criterion, we decided that our code needed to be more efficient. Before, we stored all of the nodes in a heap in order to rank them. However, we figured that this was inefficient, as popping and pushing onto a heap required $O(\log n)$ access. Instead, we decided to use a min-heap of size 10 so we would only ever keep track of the top 10 nodes. This made a slight improvement, but still not enough to get us sub-30 with the penalties.

10.

02/15/13 0:28:17 + 0:02:00 penalty = 0:30:17	*Stopping criterion*: Stop after 9 iterations *Tasks*: 20:10 PageRank, 1:1 process *Changes*: No tracking of relative ranks

We changed the stopping criterion from tracking the stability of the top 10 ranks to simply stopping after a fixed number of iterations. Our choice of 9 iterations was somewhat arbitrary, as we had not tried this stopping criterion before, and had no baseline to compare times to. The runtime decreased slightly, but the penalty increased, so the improvement was not substantial.

11.

02/15/13 0:31:28 + 0:24:30 penalty = 0:55:58	*Stopping criterion*: Stop after 5 iterations *Tasks*: 20:10 PageRank, 1:1 process *Changes*: Stopping after 5 iterations instead of 9

Since even our very fast and strict stopping criterion still was not fast enough, we decided to take a risk and have the algorithm stop after 5 iterations. Oops! This was a bad guess, as we stopped extremely early and resulted with a large penalty. What is strange though is that it took longer than running 9 iterations!

4 Testing

4.1 Local Testing

4.1.1 Test Automation

To run our pagerank algorithm locally, we created a bash script to iterate through each run, instead of having to manually type up the command each time.

```
#!/bin/bash
for ((i = 1; i <= $1; i++))
do
  echo Step $i
  python pagerank-map.py < output | sort -k 1,1 | python pagerank-reduce.py \
    | python process-map.py | sort -k 1,1 | python process-reduce.py > output;
done
```

4.1.2 Erdős-Rényi Graphs

In order to test our algorithm on more data sets, we tried to generate random directed graphs. Our first model was the Erdős-Rényi model. As discussed in class, this model starts with n vertices, and forms any possible undirected edge with probability p. This was done using `ErdosRenyiGenerator` in APGL (Another Python Graph Library). Then, to make the edges directed, we made the edge go one way with probability p, the other way with probability q, and both ways with probability $1 - p - q$.

However, when we tried running our PageRank algorithm on random Erdős-Rényi graphs with several thousand nodes, the algorithm converged in every case in no more than 4 iterations. This contrasted the random Erdős-Rényi graph provided to us with only 100 nodes, which converged in 29 iterations. We realized that for very high number of vertices, the degree distribution of the graph would not be heavy-tailed, meaning that this model was not very accurate, and therefore not very good to use for testing.

4.1.3 Barabási-Albert Graphs

After some research, we found a better model to generate graphs with heavy-tailed degree distributions, the Barabási-Albert Model. This model generated graphs with heavy-tailed degree distributions using preferential attachment, which we also discussed in class: starting with a central node, add nodes to the graph one at a time, attaching edges preferentially to nodes with higher degree. We did this with `BarabasiAlbertGenerator` APGL. We directed all edges from newly added nodes toward the existing nodes. Figure 6 shows a graph generated with model with 1000 nodes.

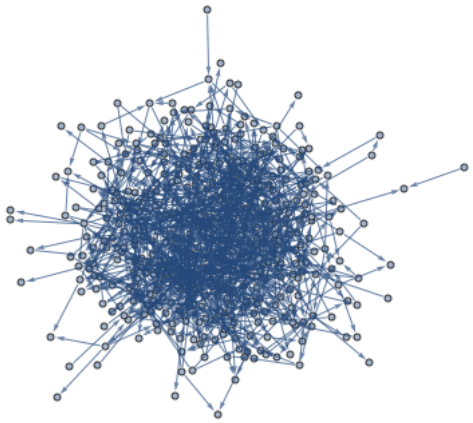


Figure 5: *Graph of the connections between* 500 *nodes in an Erdős-Rényi graph* ($p = 0.01$).

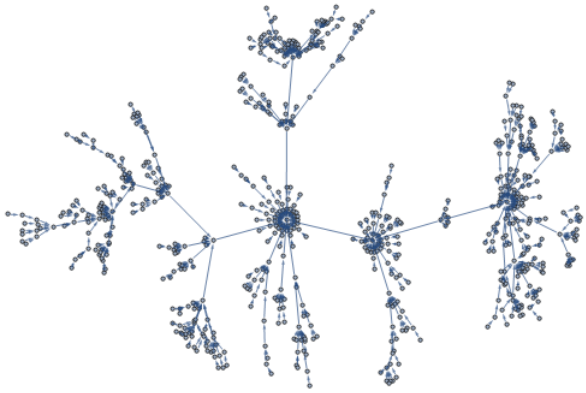


Figure 6: *Graph of the connections between* 1000 *nodes in a Barabási-Albert graph.*

4.2 Testing on Amazon

4.2.1 Automated Script

We also wrote a script `script.py` that would automate sending jobs to Amazon EC2. This script included the uploading and downloading of files.

4.2.2 Throttling

A problem that multiple teams had when testing on Amazon EC2 was throttling. This occurred when polling Amazon for the job status, as well as anything that involved jobs, such as submitting and adding steps. To solve this problem, we wrote a helper method that accepted a function pointer as an input. If any throttling occurred, the method would try running the function again until it succeeded.

```
def run_cmd(func, args, action, sleep_time=10):
    """Runs the command, but handles throttling by Amazon.

    If throttling occurs, sleeps for a specified time (default 10s).

    Args:
        func        Function to call on.
        args        Arguments for the function call, as a tuple.
        action      Line to print out.
        sleep_time  Length of time in seconds to sleep if throttling occurs.
    """
    while True:
        try:
            func(*args)
            break
        except EmrResponseError:
            print action
            time.sleep(sleep_time)
```

5 Conclusion

5.1 Final Results

Once we implemented the basic algorithm, our team focused primarily on optimizing the stop criterion to improve our rankings. The three main aspects we considered were the maximum difference in PageRank, the changes in relative ranks, and the number of iterations. We found that stopping after the relative ranking of the top n nodes stabilized was a more efficient method than waiting for the sum of the residuals to become arbitrarily low, since it provided a mostly-correct result for significantly fewer iterations. This method was also superior to choosing an arbitrary number of iterations, since it took advantage of the heavy-tailed properties of the graph, and is thus more algorithmically sound.

5.2 Suggestions for Next Year

There are a few suggestions that our team has on how to improve the Rankmaniac assignment for next year:

- Provide students with more information about the Amazon setup. We spent some time debugging due to Amazon using Python 2.5, and not knowing that Amazon chooses the number of map tasks based on the input data, while the values set in the `num_map` and `num_reduce` variables are only interpreted as lower bounds.
- Access to the Bash or Python script that uploads code to Amazon. This makes testing easier so that we can focus on the algorithm rather than setting up on EC2.

- Stricter penalties. The final round of runs involved teams just trying to reduce the time of the algorithm with as little penalty as possible. The fastest way to do so, directly limiting the number of iterations, no longer follows the MapReduce paradigm, which is somewhat against the purpose of the assignment. It also tailors the algorithm to the specific dataset, which is inappropriate.

- More representative grading datasets. Based on the pattern of parameter choices and aesults from our Amazon runs, we conjectured that one dataset used for grading converged rather quickly and was likely generated from an Erdős-Rényi random graph, which is not heavy-tailed. The second dataset, on the other hand, appeared to be heavy-tailed but did not contain enough pathological features to really exercise our algorithms.

6 References

[1] A. Sarma, A. Molla, G. Pandurangan, and E. Upfal. Fast Distributed PageRank Computation. *arXiv:1208:3071*, 2012.

[2] G. Del Corso, A. Gulli, and F. Romani. Fast PageRank Computation via a Sparse Linear System. *Internet Mathematica*, vol. 2, no. 3, pp. 251-273, Aug. 2005.

[3] H. Ishii and R. Tempo. Distributed Randomized Algorithms for the PageRank Computation. *IEEE Trans. Autom. Control*, vol. 55, no. 9, pp. 1987-2002, Sep. 2010.

[4] T. Haveliwala. Efficient Computation of PageRank. *Stanford University*, Oct. 1999.

[5] Y. Chen, Q. Gan, and T. Suel. I/O-Efficient Techniques for Computing Pagerank. *CIS Dept., Polytechnic Univ.*, 2002.

[6] P. Boldi, M. Santini, and S. Vigna. PageRank as a Function of the Damping Factor. *Universitá degli Studi di Milano*, 2006.