Networks: Structure and Economics CMS/CS/EE/IDS 144
Miniproject 1: Rankmaniac February 18^{th}, 2020

Group: BirbNet

---

1 Team Contributions

- **Tanvi:** Participated in brainstorming sessions. Pair programmed initial algorithm. Debugged initial algorithm.
- **Ziyan:** Participated in brainstorming sessions. Pair programmed initial algorithm. Debugged initial algorithm.
- **Maria:** Participated in brainstorming sessions. Pair programmed initial algorithm.

2 Algorithm Overview

PageRank The core implementation of Google's PageRank is summarized by the equation:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

where each p_i is a page (node in the web graph), d is a damping factor, $M(p_i)$ is the set of pages linked to from p_i, and $L(p_j)$ is the number of outgoing links from page p_j. Iterative PageRank updates each page's probability on each pass as a weighted sum of contributions from all the pages that point to it. In this version, the damping factor d allows us to account for the scenario that no outlinks from a particular page are followed (i.e. an imaginary "web surfer" stops clicking through on their random walk). The web graph's adjacency matrix can also be used to compute an algebraic version of PageRank.

MapReduce The two key steps of the parallelized MapReduce implementation are "Map", which applies a function to each `<key, value>` pair passed in, and "Reduce", which applies a function to the (hopefully) collated `<key, value>` pairs that have been aggregated by the collector.

Implementation Here, we use MapReduce to implement iterative PageRank. Each iteration consists of two consecutive Map/Reduce steps; the first one is "pagerank map/reduce" and the second "process map/reduce". Each of these steps functions as follows:

- **`pagerank_map`** Takes as input the raw data in `NodeId:01̂.0,0.0,83,212,302` form, and distributes the current node's current score equally among each of the nodes it has outgoing links to (the edge list). For iterations after the first, each of these input lines also contains the iteration number. It outputs one line with the distributed amount as new score for each of the nodes in the edge list, and an updated line for the current node with the distributed amount deducted from the current score. In order to support early stopping, some of the iterations might also send across a line with the top 20 nodes in order along with a special delimiter - we simply pass this line through to the next function.
- **`pagerank_reduce`** The MapReduce "collector", which in this case simply sorts by key, allows us to collect a list of score fractions for each node, as well as a list of adjacent nodes. This function simply computes the total score and adjacency list for each node and returns these, along with the extra information (iteration number, etc). Note that the score for each node is calculated according to:

  $$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} + C$$

  where $d = 0.85$ is the standard damping factor and $C = 0.2$ is an inflation factor (discussed further in optimizations).
- **`process_map`** This function simply serves as an identity function to filter through our updated nodes.

- **`process_reduce`** This function takes in the updated nodes and, starting at iteration 8, checks for early termination every few iterations based on convergence. In order to accomplish this, we compare against the previously set up list of top 20 nodes. We return updated scores and edge lists for each node in initial input format for the next MapReduce iteration.

3 Optimizations

We started with the basic PageRank algorithm described in the previous section, without early stopping. This resulted in completely correct ranking but a 5 hour run time. However, when looking at the output of the smaller data sets, we found that our algorithm converged to the correct ranking of the top 20 nodes after about 16 iterations. Thus, we added a check for convergence after iteration 14, and continued checking for convergence after every 2 iterations. We tested different numbers for the number of iterations to wait between convergence tests, and found that testing every iteration gave false positives, as it was possible that the rankings would seem to converge to incorrect values early on and then change. Checking every 3-5 iterations also worked, but we felt the small increase in accuracy was not worth the longer runtime of those extra iterations. We chose to check for convergence only after iteration 8 since that is where the ranks seemed to stabilize in our test dataset, and we wanted to minimize the runtime overhead of costly comparison operations that would have to be performed to test for convergence. For our implementation, ranking information was stored as a single line with the nodes separated by a different delimiter than regular nodes, and passed through the different MapReduce calls in the same manner as the other data (since there was no way to store intermediate data).

We were originally calculating N, the total number of nodes in the graph, in the `process_reduce` step, where we were sure we would have access to all of the nodes concurrently. This calculated value was used to perform the normalization of $1 - d$ as described in the formula presented above, and was then passed as a parameter to use in future iterations. Note that this necessitated appending N to each line of data that we output from `process_reduce`. However, this run took 2:04 min, and in testing our accuracy on the local dataset with and without normalizing with N, we found that our accuracy was not significantly worsened without normalization. By removing the calculation of N and the parsing of N in each of the methods, we were able to reduce our algorithm runtime from 2:05 to 1:55 without impacting the accuracy of our rankings. This also resulted in a 3.9 MB to 2.9 MB file size decrease per iteration for the largest local test dataset, which no doubt helped with the faster run time.

Using N alone also resulted in increasingly smaller weights - with a large enough graph, 1/N approaches 0. Simply scaling the score by 1.1 did not yield correct results; since 1/N approaches 0, this becomes the naive PageRank algorithm that is directly proportional to degree. To counteract the tendency for scores to become vanishingly small, we tried adding a constant factor. Since the score is decreasing by about 15% each iteration, we tried an inflation of 0.2 of the score per node per iteration. In combination with early stopping, this yielded surprisingly good results after 8 iterations.

We tried to keep to native Python methods whenever possible to preserve the built-in optimizations. For example, we used the "+" operator for string concatenation (it's faster that any more complicated approach we could have written ourselves).

Finally, as a quick experiment, we tried using degree centrality as a measure of ranking (since we proved in the previous set that it was proportional to naive PageRank). This would have been extremely fast computationally, since it only requires one MapReduce iteration for degree counting, but unfortunately the inaccuracy penalty was too high to make up for time saved.

4 Network Structure

We knew nothing about the network structure of the large private data set we would be tested on so we thought it would be best to use a generalizable PageRank algorithm that would perform on any dataset.

OOOOOOOooooo so we can "derive" the 0.85 alpha damping factor number because it comes from the property that the network graph is heavy tailed. This is sort of a measure of tolerance of how close it needs to be to the rankings to be right. (so pick an acceptable error and then number of iterations = acceptable error / damping factor). Also catastrophe principle? The top ranks are gonna be bigger and more distributed so there is a smaller chance of them being wrong and they will converge more quickly that the node further down in the list of ranks.

5 Future Steps

Dynamic programming? Link to papers read ¡@Tanvi¿? Calculating the full transition matrix required for this seemed like it would defeat the point of using MapReduce though, because not all computations would be parallelizable. Make it less accurate (less iterations) so that it will run faster, since the punishment for incorrect rankings is relatively small. As the number of iterations get smaller (but above 1), pagerank acts more as a degree ranker, which in itself has decent accuracy since graph degrees are a heavy tail distribution. The biggest thing slowing down our code was parsing our data from a string to list of integers in every function and we could have found a way to optimize this using more pythonic data structure. ¡put everything in a map?¿ ¡Also – might have been a better way to store data in the files. ¿