CS 144 Miniproject: Rankmaniac

Wen Gu, Sha Sha, Yu Wu

1 Introduction

1.1 PageRank

PageRank is an algorithm used to rank web pages in search engine results. It is a way of measuring the importance of website pages. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.

Consider a web graph $G(V,E)$ with n nodes. Each node represents a webpage. Then the transition matrix is

$$P=\{p_{ij}\}$$

$$p_{ij}=\begin{cases}\dfrac{1}{d_i}, & i\to j\\ 0, & otherwise\end{cases}, p_{ii}=\begin{cases}1, & d_i=0\\ 0, & d_i\neq 0\end{cases}$$

where d_i is the outdegree of node i.

In order to make PageRank converge, we use the notation $G=\alpha P+\dfrac{1-\alpha}{n}\cdot 1_{n\times n}$ as transition matrix, where α is damping factor. Let r_i denote the PageRank of page i, and $r=(r_1,r_2,\cdots,r_n)$ denote the vector of PageRanks of all pages. $r(t)$ is the vector of PageRank of all pages at iteration t. For all pages, the PageRank is initialized to $\dfrac{1}{n}$. The iteration calculation is defined as

$$r(t+1)=r(t)\cdot G$$

Another way to calculate the PageRank is to solve

$$\pi=\pi G$$

where π is the vector of PageRank of all pages.

1.2 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.

A MapReduce framework is usually composed of three steps:

- Map: each computer applies the map function to the data in parallel, and emit key-value pairs.
- Collect: computers collect the key-value pairs and redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same computer.
- Reduce: each computer processes each group of output data, per key, in parallel.

2 Basic Algorithm

We implement the PageRank Algorithm with two MapReduce processes. PageRank step calculates and updates the page rank of each node in each iteration. Then Process step determines if current result satisfies the stop condition and output the top 20 nodes.

2.1 PageRank_Map

PageRank_Map reads in the graph information, including the page rank of node i and its outgoing links j. The format of original information is: iter@NodeId:I\t current_pagerank, previous_pagerank, out_node_1, out_node_2, ... Then it emits node i's contribution to its outgoing links j, which is $c_{ij} = \begin{cases} \alpha \cdot PR_i \cdot \frac{1}{d_i}, & i \to j \\ \alpha \cdot PR_i, & d_i = 0 \end{cases}$, where PR_i is current PageRank of node i and d_i is outdegree of node i. In addition, pagerank_map also emits the original graph information for pagerank_reduce and next iteration.

PageRank Map

for line in stdin:

line = iteration, node i , curr_pagerank, prev_pagerank, listOfOutgoingLinks

for node j in listOfOutgoingLinks:

contribution $c_{ij} = \begin{cases} \alpha \cdot PR_i \cdot \frac{1}{d_i}, & i \to j \\ \alpha \cdot PR_i, & d_i = 0 \end{cases}$

stdout(iteration, node j, c_{ij})

stdout(line)

2.2 PageRank_Reduce

PageRank_Reduce first collects the information emitted by PageRank_Map and constructs a map, where the key is each node and the value is other nodes' contributions to it and the graph information as well. Then it calculates the new PageRank of each

node j by $\sum_i c_{ij} + (1-\alpha)$, updates and emits.

PageRank Reduce

Collector
Collect all outputs from Map() in form of (key, value) pairs in a new list called listOfMapOutputs:

(node i, {c_{ij}, {iteration, node i, curr_pr, prev_pr, listOfOutgoingLinks}})

# Reducer
for node i in listOfMapOutputs:
    prev_pagerank = curr_pagerank
    curr_pagerank = $\sum_i c_{ij} + (1-\alpha)$
    stdout(iteration, node i, curr_pr, prev_pr, listOfOutgoingLinks)

2.3 Process_Map

Process_Map just outputs the original information as it comes in.

Process Map

```
for line in stdin:
    stdout(line)
```

2.4 Process_Reduce

Process_Reduce collects the information emitted by PageRank_Reduce and checks if current PageRank satisfies the stop condition. If stop condition is satisfied, it outputs the PageRanks and the node id of top 20 nodes and stop running program. Otherwise, it outputs the original line and starts next iteration.

Process Reduce

for line in stdin:
    line = iteration, node i, curr_pagerank, prev_pagerank, listOfOutgoingLinks
    curr_pr_heapq ← (node i, curr_pagerank)
    prev_pr_heapq ← (node i, prev_pagerank)
    record ← line

for i = [1, 20]:

$$a_i = \frac{|curr\_pr\_heapq_i - prev\_pr\_heapq_i|}{prev\_pr\_heapq_i}$$

if $\frac{1}{20}\sum_{i=1}^{20} a_i \leq threshold$:

    for i = [1, 20]:

        stdout($curr\_pr\_heapq_i$, node i)

else:

    for line in record:

        stdout(line)

3 Stop Condition and Optimization

This section talks about the specific stop condition and other optimizations we have tried and how we choose the final method. The optimization involves the tradeoff between running time and the accuracy which can be represented in following formula:

$$argmin\ T_{running} + \sum_{i \in S} 30 * |f(i) - r(i)|^2$$

where $T_{running}$ is the running time of algorithm in sec, S is the set of nodes ranked in top 20 calculated by our algorithm, $f(i)$ is the calculated ranking order of node i, and $r(i)$ is the exact ranking order.

3.1 Stop with Strict Condition

When applying iterative method, we calculate $r(t+1) = r(t) \cdot G$ every iteration. Once $r(t+1) = r(t)$, the vector of PageRanks of all the pages will no longer change. Therefore, one strict stop condition is that current PageRanks values are the same as previous PageRanks values. However, although this strict condition will give us the exact ranking of each node, but it takes very long time to converge. Running on AWS, it takes 50 iterations (reach the maximum iteration allowed), which takes five and a half hours. Therefore, optimizations are required to improve the performance.

3.2 Stop with Top k Order Condition

Since our final goal is to find top 20 pages in order and we do not care about the value of PageRanks, we could relax the condition by stopping iteration when the current relative order of top 20 pages is the same as previous relative order of top 20 pages no matter if their PageRank values are the same.

Since the network degree distribution is heavy tailed, the PageRank values of top k pages are much greater than those of the rest pages, which means they could converge more quickly. Therefore, we can stop iteration earlier by only considering the top k pages. Although the result may not as accurate as before, it is acceptable as long as the penalty is relatively small. With $k = 20$, the running time is about one and a half hours and the penalty is six minutes, which is a great improvement.

3.3 Stop with Relative Difference Condition

We tried to further fasten the calculation at the expense of correct ordering of top 20 pages. The iteration stops when the average relative difference of current PageRank values and previous PageRank values of top 20 pages is less than threshold. The relative difference of top i^{th} page can be calculated by

$$a_i = \frac{|curr\_pr_i - prev\_pr_i|}{prev\_pr_i}$$

When $\frac{1}{20}\sum_{i=1}^{20} a_i \le threshold$, the iteration stops. The key is to select a proper threshold since a larger threshold may lead to a greater order error while a smaller one may cause a longer running time. We tried several threshold values to benchmark the tradeoff between accuracy and speed. When test the data locally, the result is shown in the table, where iteration represents the running time to some degree because the running time of each iteration should be similar.

The following result is based on test set Gnutella.

Threshold	Number of mis-ordered pages in top 20	Iteration
0.1	12	3
0.05	6	4
0.02	4	5
0.01	4	5

Note that we use the number of mis-ordered pages in top 20 instead of official squared penalty for error. The reason is that the team does not know the exact ranking of some mis-ranked top 20 pages, which should not have been ranked in top 20.

After testing on AWS, we finally choose threshold to be 0.01, and the running time is one hour and ten minutes with the penalty of thirteen minutes, which is better than Top k order condition.

This approach gives us the best result in terms of trade-off between accuracy and running time, and thus turns into the final implementation.

3.4 Top α Percent Graph Reduction

What's more, we also tried graph reduction method. The detailed pseudo code is shown as following.

Top α Percent Graph Reduction

```
Initialize 0< α <1
Initialize M: minimum number of pages in graph
Initialize G: graph G
for each iteration:
    PR = Calculated the Pagerank of graph G
    Rank = Get the ranking order of all node according to PR
    Sorted_Rank = sort(rank)
    k = int (α * number of pages in G)
    S = set of top k ranked node in Sorted_Rank
    if number of node in k<= M:
        if previous order of rank == current order of rank
            return result
    else:
        reconstruct G based on nodes in S, maintain the edges between node in S
```

For every iteration, we first calculated the PageRanks of all pages, then sort the pages by their PageRanks. With ranking of the pages, we select top α percent of nodes with highest ranking to represent the new graph. The iterations keep going until the number of nodes is smaller than threshold value M. When the number of nodes is smaller than threshold M, we no longer reduce the graph. With graph of size M (or slightly smaller than M), the PageRanks calculation keeps going until ranking order from last iteration is equivalent to ranking order in current iteration. This method is intuitive because of 'winner takes all' effect: the pages with greater in-degree would have a faster growth in Pageranks values. Every iteration, we eliminate the weaker competitor (ranks at bottom), and only take competitive 'pages' (top α Percent) to represent the new graph. In this way, with the number of pages decreases, the PageRank calculation will be faster in every iteration.

We have tried this method locally, and it turns out that it reaches our expectation in terms of speed boosting. However, it has poor accuracy: this method selects most of top 20 pages accurately but ranks them in a wrong order. This might be due to the

sparsity of the graph. Loss of links may lead to loss of flows between pages, and thus the PageRank calculation would not be precise.

4 Conclusion

In this project, we used MapReduce to implement PageRank Algorithm to obtain the top 20 pages. The algorithm was implemented with two sequential MapReduce processes. PageRank step calculates and updates the page rank of each node in each iteration. Then Process step determines if current result satisfies the stop condition and output the top 20 nodes.

We first tried strict stop condition, which requires the current PageRanks exactly equal to previous ones. But it takes too long to reach the stop condition. Therefore, we tried to reduce the running time by tolerating more ordering errors. Both top k order condition and relative difference condition, which follow the trade-off between accuracy and speed, performed much better than the strict condition. As a result, we achieved the second place on the score board.

For future improvement, we may consider clustering algorithm so as to significantly reduce the calculation but with fair accuracy. What's more, Yasuhiro and et al. proposed a fast algorithm for exact top-k page rank in [1] by converging the upper bound and lower bound for selecting the top-k candidates. If we have more time, we would try this method to see how it performs.

5 Team Information

5.1 Team Name

SWG

5.2 Group Member

Yu Wu

Wen Gu

Sha Sha

5.3 Division of Labor

Yu Wu: collaborated on implementing basic PageRank Algorithm; optimized the stop condition; submitted the models on Amazon Sever; report writing.

Wen Gu: collaborated on implementing basic PageRank Algorithm; optimized the stop condition; collaborated on implementing optimization; report writing.

Sha Sha: optimized the stop condition; collaborated on implementing optimization; report writing.

Reference:

[1] Yasuhiro Fujiwara, Makoto Nakatsuji, 'Fast and Exact Top-k Algorithm for PageRank', Association for the Advancement of Artificial Intelligence, https://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6162/6875