# Cloud Programming HW03

Ming-Chang Chiu 100060007

June 14, 2015

## 1 Introduction

In this assignment, we are required to implement PageRank algorithm to construct a search engine which can prioritize the related links when we type in a keyword.

## 2 System Details: Algorithms

In total, I wrote four MapReduce phases to complete this assignment, including graph building, inverted index building, rank computing, result sorting. In addition, I implement four HDFS, HBase manipulation programs to help me get the desired output.

As for the implementation of PageRank on MapReduce framework, I simply follow the flow as follow:

```
1: class Mapper
2:     method Map(nid $n$, node $N$)
3:         $p \leftarrow N$.PageRank/$|N$.AdjacencyList$|$
4:         Emit(nid $n, N$)                              ▷ Pass along graph structure
5:         for all nodeid $m \in N$.AdjacencyList do
6:             Emit(nid $m, p$)                          ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid $m, [p_1, p_2, \ldots]$)
3:         $M \leftarrow \emptyset$
4:         for all $p \in$ counts $[p_1, p_2, \ldots]$ do
5:             if IsNode($p$) then
6:                 $M \leftarrow p$                      ▷ Recover graph structure
7:             else
8:                 $s \leftarrow s + p$                  ▷ Sums incoming PageRank contributions
9:         $M$.PageRank $\leftarrow s$
10:        Emit(nid $m$, node $M$)
```

The tricky part is, when dealing with dangling node, we could have difficulty broadcasting the sum of the score of all dangling nodes to all other running tasks. My solution is, first we create a file, say, info.txt to store the sum of dangling nodes and the total page number which the input file has, and when we need the information of them, we then just read the info.txt and then do the calculation. And after the iteration, we shall update the info.txt with newest sum of dangling nodes.

One trick worths mentioning is, when both killing non-existent nodes and getting sum of dangling nodes, we can attach a symbol as the node's value. By recognizing the symbol in the later phases, we can easily discern the status of the node, and the do the appropriate processes.

## 3 Usage

Basically, just execute scripts that I wrote and then one can acquire the desired output. Please follow the order below:

**invertedindex.sh:** Build the inverted index for each word in the input's ¡text¿ tag.

**graph.sh:** Build the interconnected graph for page titles.

**clean.sh:** Clean the non-existent pages in the graph.

**rank.sh:** Perform the PageRank algorithm.

**sort.sh:** Get the output of PageRank and then output the ¡Page, Score¿ pairs sorted by score.

**upload.sh:** Load the inverted index and sorted PageRank to HBase. Tables are named as `100060007_invertedindex`, `100060007_pagerank`.

**query.sh:** This is the search engine that can accept keyword. The engine will first access the invertedindex in HBase, get the corresponding pages, and then get the rank for each page, and then finally print the pages sorted by rank score.

**Notice:** In order to run correctly on each directory, one should check those 4 .sh files for the directory names. And one should modify the retrieval/ReadFile.java for the HDFS paths. The above description is based on my working directory on Quanta006.

## 4 Experiment

I perform PageRank over the 100M input file and the following is the result I got.

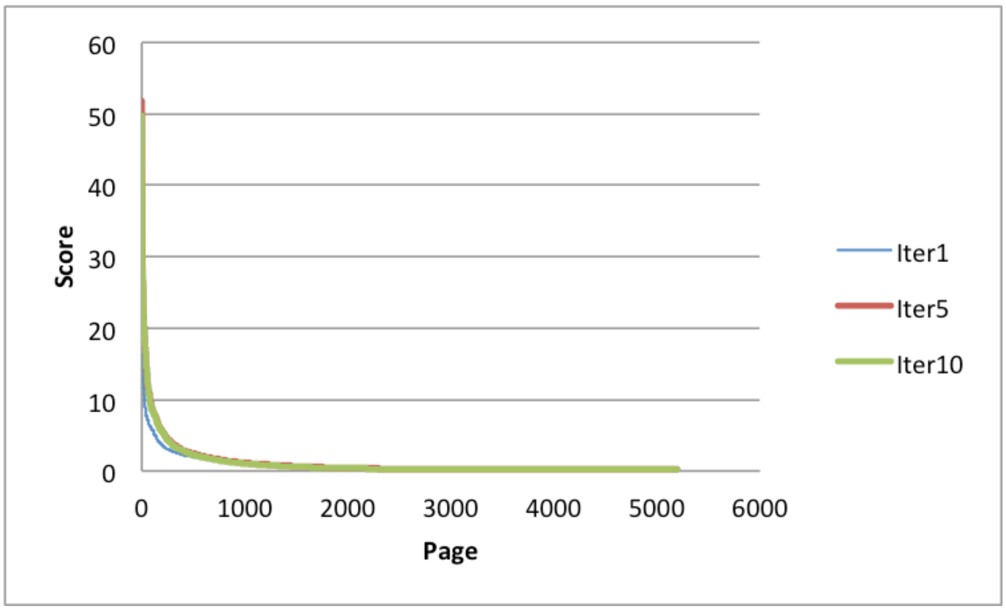

Figure 1: Distribution of score

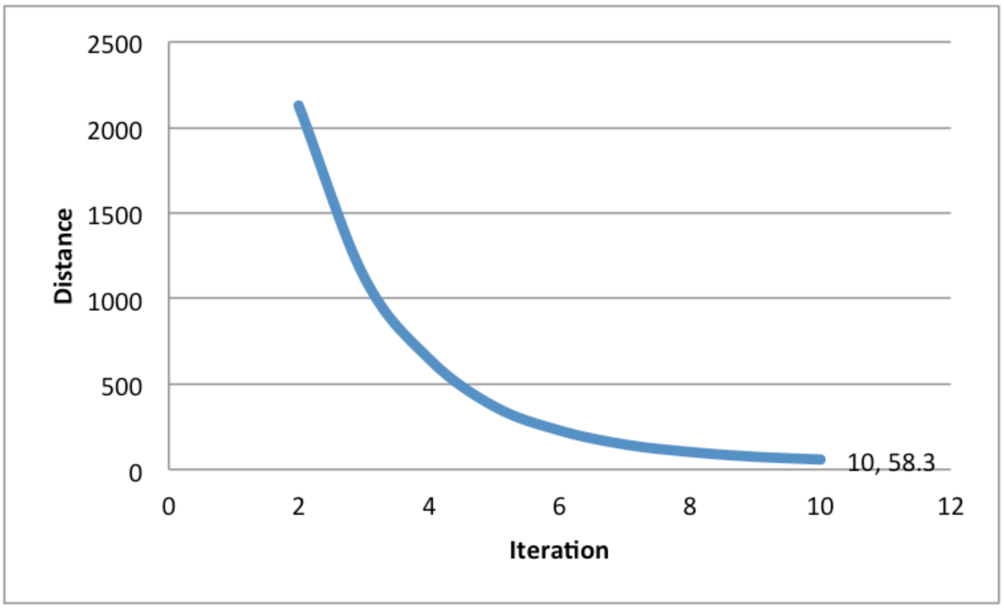

Figure 2: Distance

## 5 Observation

In general, from $figure1$, we can see that the distribution of rank score will decay exponentially. The distribution of iteration 5 and 10 are really close, which can be a sign of convergence. From $figure2$, we see that the distance will also exponentially decline, which means the scores will fast be convergent after not too many iteration, where the definition of distance is defined as $\sum_{i=1}^{N} |PR_i - PR_{i-1}|$.