# Homework4

Name: Yang Cai

## Program Discussion

### Transformation of the data

Step 1 reads input file: SparkContext. textFile: String => RDD[String]

Step 2 parses each line to adjacent list:
flatMap: RDD[String] => RDD[(String, java.util.List<String>)

Step 3 converts from Java list to Scala list and remove self-links:
map: RDD[(String, java.util.List<String>) => RDD[(String, List[String])

Step 4 merges all pairs that have same key, generate and persist this graph
reduceByKey: RDD[(String, List[String]) => RDD[(String, List[String])

Step 5 count the number of nodes: graph.count: RDD[(String, List[String]) => Int

Step 6 generates the initial page rank value:
mapValues: RDD[(String, List[String]) => RDD[(String, Double)]

Loop starts:
Step 7 joins the graph and the pagerank:
join: RDD[(String, List[String]),  RDD[(String, Double)] => RDD[(String, (List[String], Double))

Step 8 emits each page's contribution to its out-link, meanwhile add dangling nodes' rank to the global counter: flatMap: RDD[(String, (List[String], Double)) => RDD[(String, Double)]

Step 9 aggregates all contributions to the same page
foldByKey: RDD[(String, Double)] => RDD[(String, Double)]

Step 10 retrieves the dangling nodes' rank from global counter, use RDD.count to force an action(otherwise global counter won't be retrieved from driver grogram)

Step 11 calculate the final page rank value according to the formula and dangling nodes' rank.
mapValues: RDD[(String, Double)] => RDD[(String, Double)]
Loop Ends
Repeat Step 7 to Step 11 10 times

Step 12 take top 100 pages: takeOrdered: RDD[(String, Double)] => Array[(String, Double)]

Step 13 save this final output to text file: parallelize and saveAsTextFile:
Array[(String, Double)] => RDD[(String, Double)] => save to file

### Narrow/Wide Steps:

Narrow steps are, Step 2, Step 3, Step 5, Step 6, Step 7, Step 8, Step 10, Step 11, Step 12
Wide steps are Step 4, Step 9
There are total 24 stages: 1 for Step 1(SparkContext. textFile), 1 for Step 4(reduceByKey), 20 for the 10-times loop(each loop has 2 stages, the i'th loop foldByKey, and also the i – 1' th loop's foldByKey is also counted), 1 when calling takeOrdered(this triggers a stage since it needs result of foldByKey of the 10th loop), and 1 for Step 13(parallelize)

## Performance Comparison

1. Hadoop 5 workers: 4394 seconds
2. Hadoop 10 workers: 2762 seconds
3. Spark 5 workers: 2570 seconds
4. Spark 10 workers: 1228 seconds

The Spark is relative faster, it might because Spark does not read/write from/to HDFS again and again, so it has smaller I/O overhead and everything is in memory.