Assignment 4

The toy dataset is the following graph. The PageRank values are already known. We can use it to check your program.

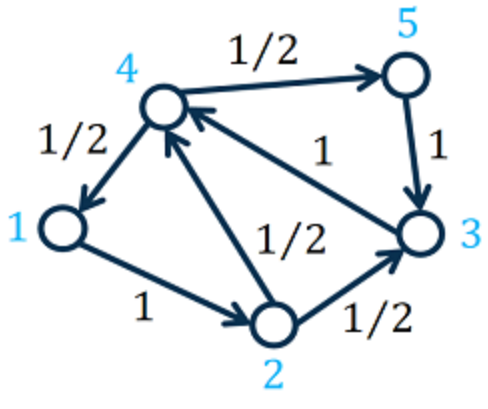

Figure 1: A toy graph for computing PageRank. The number on the edge represents the transition probability from one node to another.
The PageRank values are given in the following table (given that the decay factor):

Nodes	PageRankValues
1	0.1556
2	0.1622
3	0.2312
4	0.2955
5	0.1556

PageRank:
Compute the PageRank value of each node in the graph. Please refer to the slides for more details about the PageRank method. The key PageRank equation is as follows.

where represents the PageRank vector with each element representing the PageRank value of node , represents the number of nodes in the graph, represents the transition probability matrix with each element representing the transition probability from node to node , represents the degree of node , represents the transpose of , represents a decay factor, represents a vector of all 1's, and represents the number of nodes in the graph.
Please see the slides for more details.
In this assignment, we set the decay factor and set the number of iterations to 30.
Implementation:
Design and implement a PySpark program to compute the PageRank values. A template "PageRank_Spark_Incomplete.py" file is given.
Requirements:
You will notice that in PageRank_Spark_Incomplete.py, will have the RDD like below:
AdjList1 = ['1 2', '2 3 4', '3 4', '4 1 5', '5 3']
PageRankValue = [(1, 0.2), (2, 0.2), (3, 0.2), (4, 0.2), (5, 0.2)]

You must use RDD operators to perform transformations and actions with these 2 RDD only.
For any output of your code that is not performed using RDD operators, you will lose 10 points

The outputs in the terminal of the ground-truth solutions is given in the file "TerminalOutputs.txt". You may use it to verify your solution.
Example command to run the ".py" file:

```
spark-submit  PageRank_Spark_Incomplete.py
```

The files can be put in local file system.

Report

Please write a report illustrating your experiments. You need to explain your basic idea about how to design the computing algorithm. You may add comments to the source code such that the source code can be read and understood by the

graders.
In the report, you should include the answers to the following questions.

1. Explanation of the source code:
   What are the functions of your lambda functions? Which kind of intermediate results are generated?

   The program is broken down into 3 functions.

   - Mapper Function
   - Calculate PageRank function
   - Output function

   ▼ **The mapper function**

```
def mapper():
    # Load the adjacency list file
    AdjList1 = sc.textFile("/home/nur_haque/02AdjacencyList.txt")
    # Initialize each page's rank; since we use mapValues,
    # the resulting RDD will have the same partitioner as links
    PageRankValues = sc.parallelize([(1,0.2),(2,0.2),(3,0.2),(4,0.2),(5,0.2)])
    PRvalues = PageRankValues.map(lambda x: x[1]).collect()
    # Geting the all the page neightbors
    AdjList2 = AdjList1.flatMap(lambda x: x[3:]).filter(lambda x: x.replace(" ", "")).collect()
    shuffeledData = []
    # Calculating the probobility
    for row in AdjList1.collect():
        it = 0
        actSze = len(row)/2
        if actSze > 2:
            shuffeledData.append(1/(actSze-1))
            shuffeledData.append(1/(actSze-1))
        else:
            shuffeledData.append(1/(actSze-1))
        it+=1
    PageRankValues = sc.parallelize(shuffeledData).collect()
    # creating the mapper matrix by combining PageRanks, and Probability
    mapperMatrix = list(zip(AdjList2,PageRankValues))
    mapperMatrixRDD = sc.parallelize(mapperMatrix)
    sortedMatrixRDD = mapperMatrixRDD.reduceByKey(lambda x,y: (x,y)).sortByKey()
    return sortedMatrixRDD, PRvalues, AdjList1
```

The mapper function is designed to take the AdjList and PageRank values and return a list containing the page rank their probability and the pages they are adjacent to.

This function will return the following

```
[('1', 0.5), ('2', 1.0), ('3', (0.5, 1.0)), ('4', (0.5, 1.0)), ('5', 0.5)]
[0.2, 0.2, 0.2, 0.2, 0.2]
[' 1 2', ' 2 3 4', ' 3 4', ' 4 1 5', ' 5 3']
```

The first list is the probability of each page.

The second list just returns the probability

The third list return the normal adjacent list.

1. First the function get the all adjacent nodes

```
AdjList2 = AdjList1.flatMap(lambda x: x[3:]).filter(lambda x: x.replace(" ", "")).collect()
AdjList2 => ['2', '3', '4', '4', '1', '5', '3']
```

2. Next we calculate the probability per page based on the size of each row

```
actSze = len(row)/2
if actSze > 2:
   shuffeledData.append(1/(actSze-1))
   shuffeledData.append(1/(actSze-1))
else:
    shuffeledData.append(1/(actSze-1))
Shuffel Data => [1.0, 0.5, 0.5, 1.0, 0.5, 0.5, 1.0]
```

The logic behind the calculation is

because of the spaces we I wanted to start with half the row length.

Now I know if the length is 4 and divide by 2 I would get 2 items in that row

If I know the length is 6 and divide by 2 I would have 2 items in that row

```
actSze = len(row)/2
```

Now to compute the probability I would like to actSize -1

So if actSze is 2 so that would be 2-1 = 1

So the probability would = 1/(actSze -1) = 1/(2-1) = 1

If the actSize = 6 so that would be 3-1 = 1

So the probability would = 1/(actSze -1) = 1/(3-1) = 1/2 = 0.5

```
# Placed in a list
PageRankValues => [1.0, 0.5, 0.5, 1.0, 0.5, 0.5, 1.0]
```

```
# Now I zipped the adjacent list with the list of probability
mapperMatrix = list(zip(AdjList2,PageRankValues))
mapperMatrix => [('2', 1.0), ('3', 0.5), ('4', 0.5), ('4', 1.0), ('1', 0.5), ('5', 0.5), ('3', 1.0)]
```

```
# Now we reduce by key and and sort the data
sortedMatrixRDD = mapperMatrixRDD.reduceByKey(lambda x,y: (x,y)).sortByKey()
sortedMatrixRDD => [('1', 0.5), ('2', 1.0), ('3', (0.5, 1.0)), ('4', (0.5, 1.0)), ('5', 0.5)]
```

▼ Calculate PageRank function

```
def calcPageRank(sortedMatrixRDD, rate):
    # This hashmap depict which page/pages are impacting nth page's rank
    #  For Exmple PageRank 1 is baing impacted by 4
    # 1 -> 4
    # 2 -> 1
    # 3 -> [2,5]
    # 4 -> [2,3]
    # 5 -> 4
    mappingPR_Rate = dict({
        0 : rate[3],
        1 : rate[0],
        2 : [rate[1], rate[4]],
        3 : [rate[1], rate[2]],
        4 : rate[3]
    })
    newPageRank = []
    try:
        pageRank = sortedMatrixRDD.map(lambda x: x[1]).collect()
    except:
        pageRank = sortedMatrixRDD
    for PRs in range(len(pageRank)):
        if type(pageRank[PRs][1]) == float:
            finalVal = round((0.85 * mappingPR_Rate.get(PRs) * pageRank[PRs][1]) + ((1 - 0.85)/5),3);
            newPageRank.append(finalVal)
        elif type(pageRank[PRs][1]) == tuple:
            finalVal = round((0.85 * mappingPR_Rate.get(PRs)[0] * pageRank[PRs][1][0]) +
            (0.85 * mappingPR_Rate.get(PRs)[1] * pageRank[PRs][1][1]) + ((1 - 0.85)/5),3);
            newPageRank.append(finalVal)
    return newPageRank
```

The calcPage rank is used to compute the page ranks which is stored in newPageRank, which replaces the rate with each iteration.

But there is problem the the when calculating the page rank for the first iteration the the function woks fine. The second iteration onward it would now work because the rate have changed. So we need to get some way to get the page rank value that the nth page is being impacted by. The hashmap is used map these values correctly by using key/value paist. where the value is being indexed from rate parameter.

```
mappingPR_Rate = dict({
        0 : rate[3],
        1 : rate[0],
        2 : [rate[1], rate[4]],
        3 : [rate[1], rate[2]],
```

```
        4 : rate[3]
    })
```

This following will depict each page the pages that impacting its page rank

```
# 1 -> 4
# 2 -> 1
# 3 -> [2,5]
# 4 -> [2,3]
# 5 -> 4
```

▼ The output function

This function runs through the 30 iteration and prints the page ranks for each iteration

```
def output():
    sortedMatrixRDD = mapper()[0].collect()
    rate = mapper()[1]
    # Run 30 iterations
    print ("Run 30 Iterations")
    for i in range(1, 30):
        print ("Number of Iterations")
        sortedMatrixRDDCalc = calcPageRank(sortedMatrixRDD, rate)
        print("========================================")
        print (str(i) + " => " + str(sortedMatrixRDDCalc))
        print("========================================")
        # print(sortedMatrixRDDCalc)
        rate = sortedMatrixRDDCalc
    print ("=== Final PageRankValues ===")
    print (sortedMatrixRDDCalc)
    # Write out the final ranks
    sc.parallelize(sortedMatrixRDDCalc).coalesce(1).saveAsTextFile("/home/nur_haque/PageRankValues_Final")
```

2. Experimental Results

   2.1) Screenshots of the key steps. For example, the screenshot for the outputs in the terminal when you run the command. It will demonstrate that your program has no bug.

   a. The key steps are listed above

   2.2) Explain your results. Does your implementation give the exact PageRank values?

   The numbers return exact page rank

```
Run 30 Iterations
Number of Iterations
========================================
1 => [0.115, 0.2, 0.285, 0.285, 0.115]
========================================
Number of Iterations
========================================
2 => [0.151, 0.128, 0.2128, 0.3573, 0.151]
========================================
Number of Iterations
========================================
3 => [0.182, 0.158, 0.2127, 0.2653, 0.182]
========================================
Number of Iterations
========================================
4 => [0.143, 0.185, 0.2519, 0.2779, 0.143]
========================================
Number of Iterations
========================================
5 => [0.148, 0.152, 0.2302, 0.3227, 0.148]
========================================
Number of Iterations
========================================
6 => [0.167, 0.156, 0.2204, 0.2903, 0.167]
========================================
Number of Iterations
========================================
7 => [0.153, 0.172, 0.2382, 0.2836, 0.153]
========================================
Number of Iterations
========================================
8 => [0.151, 0.16, 0.2331, 0.3056, 0.151]
========================================
Number of Iterations
========================================
9 => [0.16, 0.158, 0.2263, 0.2961, 0.16]
========================================
```

```
Number of Iterations
========================================
10 => [0.156, 0.166, 0.2331, 0.2895, 0.156]
========================================
Number of Iterations
========================================
11 => [0.153, 0.163, 0.2331, 0.2987, 0.153]
========================================
Number of Iterations
========================================
12 => [0.157, 0.16, 0.2293, 0.2974, 0.157]
========================================
Number of Iterations
========================================
13 => [0.156, 0.163, 0.2314, 0.2929, 0.156]
========================================
Number of Iterations
========================================
14 => [0.154, 0.163, 0.2319, 0.296, 0.154]
========================================
Number of Iterations
========================================
15 => [0.156, 0.161, 0.2302, 0.2964, 0.156]
========================================
Number of Iterations
========================================
16 => [0.156, 0.163, 0.231, 0.2941, 0.156]
========================================
Number of Iterations
========================================
17 => [0.155, 0.163, 0.2319, 0.2956, 0.155]
========================================
Number of Iterations
========================================
18 => [0.156, 0.162, 0.231, 0.2964, 0.156]
========================================
Number of Iterations
========================================
19 => [0.156, 0.163, 0.2314, 0.2952, 0.156]
========================================
Number of Iterations
========================================
20 => [0.155, 0.163, 0.2319, 0.296, 0.155]
========================================
Number of Iterations
========================================
21 => [0.156, 0.162, 0.231, 0.2964, 0.156]
========================================
Number of Iterations
========================================
22 => [0.156, 0.163, 0.2314, 0.2952, 0.156]
========================================
Number of Iterations
========================================
23 => [0.155, 0.163, 0.2319, 0.296, 0.155]
========================================
Number of Iterations
========================================
24 => [0.156, 0.162, 0.231, 0.2964, 0.156]
========================================
Number of Iterations
========================================
25 => [0.156, 0.163, 0.2314, 0.2952, 0.156]
========================================
Number of Iterations
========================================
26 => [0.155, 0.163, 0.2319, 0.296, 0.155]
========================================
Number of Iterations
========================================
27 => [0.156, 0.162, 0.231, 0.2964, 0.156]
========================================
Number of Iterations
========================================
28 => [0.156, 0.163, 0.2314, 0.2952, 0.156]
========================================
Number of Iterations
========================================
29 => [0.155, 0.163, 0.2319, 0.296, 0.155]
========================================
=== Final PageRankValues ===
[0.155, 0.163, 0.2319, 0.296, 0.155]
```

Code

```
import pyspark
from pyspark.context import SparkContext
from pyspark import SparkConf

conf = SparkConf()
sc = SparkContext(conf = conf)
sc.setLogLevel("ERROR")

def mapper():
    # Load the adjacency list file
    AdjList1 = sc.textFile("/home/nur_haque/02AdjacencyList.txt")
    # Initialize each page's rank; since we use mapValues,
    # the resulting RDD will have the same partitioner as links
    PageRankValues = sc.parallelize([(1,0.2),(2,0.2),(3,0.2),(4,0.2),(5,0.2)])
    PRvalues = PageRankValues.map(lambda x: x[1]).collect()
    # Geting the all the page neightbors
    AdjList2 = AdjList1.flatMap(lambda x: x[3:]).filter(lambda x: x.replace(" ", "")).collect()

    shuffeledData = []
    # Calculating the probobility
    for row in AdjList1.collect():
        it = 0
        actSze = len(row)/2
        if actSze > 2:
            shuffeledData.append(1/(actSze-1))
            shuffeledData.append(1/(actSze-1))
        else:
            shuffeledData.append(1/(actSze-1))
        it+=1
    PageRankValues = sc.parallelize(shuffeledData).collect()
    # creating the mapper matrix by combining PageRanks, and Probability
    mapperMatrix = list(zip(AdjList2,PageRankValues))
    mapperMatrixRDD = sc.parallelize(mapperMatrix)
    sortedMatrixRDD = mapperMatrixRDD.reduceByKey(lambda x,y: (x,y)).sortByKey()
    return sortedMatrixRDD, PRvalues, AdjList1

def calcPageRank(sortedMatrixRDD, rate):
    # This hashmap depict which page/pages are impacting nth page's rank
    #  For Exmple PageRank 1 is baing impacted by 4
    # 1 -> 4
    # 2 -> 1
    # 3 -> [2,5]
    # 4 -> [2,3]
    # 5 -> 4
    mappingPR_Rate = dict({
        0 : rate[3],
        1 : rate[0],
        2 : [rate[1], rate[4]],
        3 : [rate[1], rate[2]],
        4 : rate[3]
    })
    newPageRank = []
    try:
        pageRank = sortedMatrixRDD.map(lambda x: x[1]).collect()
    except:
        pageRank = sortedMatrixRDD
    for PRs in range(len(pageRank)):
        if type(pageRank[PRs][1]) == float:
            finalVal = round((0.85 * mappingPR_Rate.get(PRs) * pageRank[PRs][1]) + ((1 - 0.85)/5),3);
            newPageRank.append(finalVal)
        elif type(pageRank[PRs][1]) == tuple:
            finalVal = round((0.85 * mappingPR_Rate.get(PRs)[0] * pageRank[PRs][1][0]) +
            (0.85 * mappingPR_Rate.get(PRs)[1] * pageRank[PRs][1][1]) + ((1 - 0.85)/5),4);
            newPageRank.append(finalVal)
    return newPageRank

def output():
    sortedMatrixRDD = mapper()[0].collect()
    rate = mapper()[1]
    # Run 30 iterations
    print ("Run 30 Iterations")
    for i in range(1, 30):
        print ("Number of Iterations")
        sortedMatrixRDDCalc = calcPageRank(sortedMatrixRDD, rate)
        print("=========================================")
        print (str(i) + " => " + str(sortedMatrixRDDCalc))
        print("=========================================")
        # print(sortedMatrixRDDCalc)
        rate = sortedMatrixRDDCalc
    print ("=== Final PageRankValues ===")
    print (sortedMatrixRDDCalc)
    # Write out the final ranks
    sc.parallelize(sortedMatrixRDDCalc).coalesce(1).saveAsTextFile("/home/nur_haque/PageRankValues_Final")

def main():
    output()
if __name__ == "__main__":
    main()
```