Homework #6 – Spark Programming

Yoke Kai Wen, 2020280598

April 14, 2021

1 Introduction

In this assignment, I implemented Word Count and Page Rank algorithms with `pyspark`. The code is all written in Python using the given starter code scripts.

2 Task 1: Word Count

To implement word count in Spark, I first used regex to split the input text string by special characters (characters that are non-alphanumeric) into individual words. Then, I used the `map` function to map each word into a (word, 1) tuple, and finally used `reduceByKey` to sum up all the values belonging to each word key to get the counts RDD of each word. These steps are executed by the following line of code:

```
counts = lines.flatMap(lambda x: re.split('[^\w]+', x)).map(lambda word: (word,1))
    .reduceByKey(lambda a,b: a+b)
```

Finally, we sort `counts` in descending order and print the first five word-count pairs.

3 Task 2: Page Rank

To implement Page Rank, I first generate the `links` and `ranks` RDDs from the src-dst node pairs from the text file. The `links` RDD contains (srcNode, list of dstNodes) key-value pairs; the `ranks` RDD contains (srcNode, pageRankScore) key-value pairs. Using these intermediate RDDs, I then perform 100 iterations of the PageRank algorithm by redistributing the pageRankScore of each source node to each of its destination nodes, with a damping factor $d = 0.8$. **The total time taken for 100 iterations on the input file `full.txt` is 11.48s.**

3.1 Generating `links` RDD

The `links` RDD is generated by first splitting the input text string by the newline character which gives each src-dst node string, and then mapping each src-dst node pair into the form of (srcNode, dstNode) key-value pairs. Then, each dstNode value is transformed into a list format [dstNode] so that during reduction by key, all the dstNodes of a srcNode will be added to a list. Then, the list of dstNodes are filtered so that each dstNode appears only once since repeating edges between the same src-dst pair are counted as 1 edge only.

```
links = lines.flatMap(lambda x: x.split('\n')) \
        .map(lambda node: (node.split('\t')[0],node.split('\t')[1])) \
        .map(lambda keyval: (keyval[0], [ keyval[1] ])) \
        .reduceByKey(lambda a,b: list(set(a+b)))
```

3.2 Generating `ranks` RDD

I initialise the pageRankScores of each node uniformly, i.e. as $\frac{1}{numNodes}$. I first compute the number of unique nodes in the graph, by splitting the input text string into node IDs, and then extracting unique node IDs by mapping each node ID to (node, 1) key-value pairs and then reducing by key such that only the first occurrence of (node, 1) is retained. I then use the count method to get the number of nodes. See the line of code below for details.

```
    numNodes = lines.flatMap(lambda x: re.split('[^\w]+', x)) \
        .map(lambda node: (node,1)) \
        .reduceByKey(lambda a,b: a+b) \
        .count()
```

Then, I compute the **`ranks`** RDD, by mapping each node to (node, initial score) key-value pairs, with initial score being $\frac{1}{numNodes}$.

```
    init_score = 1/numNodes

    ranks = lines.flatMap(lambda x: re.split('[^\w]+', x)) \
        .map(lambda node: (node, init_score)) \
        .reduceByKey(lambda a,b: a)
```

3.3 PageRank iterative algorithm

Setting damping factor to 0.8, I implemented 100 iterations of the PageRank algorithm. I first join the **`links`** and **`ranks`** RDDs by their srcID keys, such that the resulting intermediate RDD contains key-value pairs of (srcID, (list of dstNodes, pageRankScore)). I then compute the contribution of srcNode to each of its dstNode by mapping the RDD into key-value pairs of (dstID, scoreContributionByASrcNode), weighted by the damping factor. I then sum up all the contributions to each dstNode using **`reduceByKey`** to get the updated pageRankScores of each node. Note that I also add a constant of $\frac{1-d}{numNodes}$ to each pageRankScore to account for damping. Furthermore, since for this assignment, there are no dead ends and all nodes are connected to each other, I did not account for dead end nodes.

```
    # damping factor d = 0.8
    d = 0.8
    for i in range(100):
        ranks = links.join(ranks) \
            .flatMap(lambda x: map(lambda dest: (dest, d* ((x[1][1] + (1-d)/
    numNodes)/ len(x[1][0]) ) ), x[1][0])) \
            .reduceByKey(lambda a,b: a+b)
    ranks = ranks.map(lambda nodescore: (nodescore[0], nodescore[1] + (1-d)/
    numNodes)) #add (1-d)/N to every node
```