Dig Data Management Assignment 8

信科四 郑元嘉 1800920541

In this report, we'll implement 'PageRank' by GraphX based on Spark.
Get it straight, we simply assign score of 1 to each nodes initially, so the sum of all nodes' scores is not equal to 1.
We will run `scala` script in `spark-shell` .
As of the network data, it's converted into structured `.txt` file from and `.png` file:

```
A C F D B
B A D G E
C A D F
D A B C E F G
E B D G
F A C D G H
G B D E F H
H F G I
I H J
J I
```

Take the 1st row as an instance, 'A C F D B' indicates node `A` is connected to nodes `C` , `F` , `D` and `B` .

Scala Script

The script is going to be broken down into segments.

```
//   val conf = new SparkConf().setAppName("pageRank")
//   val sc = new SparkContext(conf)

val fp = "/media/jack/File/hadoopHw/hw8/jack/network.txt"

var mapping = sc.textFile(fp)
.flatMap(line => line.split(" "))
.distinct
.map(node => (MurmurHash3.stringHash(node).toLong, node))
```

First things first. I'm confronted by my first bug when I'm trying to initiate `SparkContext` because the spark shell has already initiated one and the default setting allows only one `SparkContext` instance at a time.

Function `sc.textFile()` is applied here to read `.txt` file to build up one table mapping the hashed id number back into the original character name (such as A, B, ...). It returns `RDD` object instead of any related objects of `collection`, so we pay more attention when we coping with type transformation.

We use `flatMap` to extract all possible characters, use `distinct` to capture distinct ones, and eventually use `map` to create `Tuple` pairs in the favor of `MurmurHash3` 's hashing.

```
var links = List[Tuple2[Long, Long]]()

for (line <- Source.fromFile(fp).getLines) {
    var s = line.split(" ")(0)
    line.split(" ").drop(1).map(node => links = (MurmurHash3.stringHash(s).toLo
}

var linksRdd = sc.parallelize(links).persist()
```

Alternatively, we apply function `Source.fromFile()` to read and parse the `.txt` file into sourceNode-destinationNode pairs which follow the rules for the input of the function `Graph.fromEdgeTuples()` shown later. Note that due to object `Graph.Vertice`, the type of the vertices (nodes) is required as `Long` rather than default `Int` or else.

A common Trick of list is played here, we need to use `::` to add new element to the list (what's more, only add at the 'head' can be achieved), and be careful of type assignment for new empty `List` object.

To transform the pairs `List` to a `RDD`, we apply `parallelize()`. What's more, call `persist()` function to store it in RAM to speed up the task.

```
val graph = Graph.fromEdgeTuples( linksRdd, 1 )

val ranks =
    graph
    .pageRank( 0.01 )
    .vertices

val fullMap =
ranks
.join( mapping )
.map( row => row._2)
.sortBy(_._2)
.collect

```

```
15  .foreach(println)
    }
```

Function `Graph.fromEdgeTuples` is one of the ways to create one `Graph` object. We assign `1` as all vertices' attributes.
Then, we call function `pageRank` to run our page ranking task.
Lastly, we use `join()` and table `mapping` to map the hashed ids back the original node character names, and use `map()` and `sortBy()` to get a clear and wanted form of data.
Here's a point that should be kept in mind. Call `foreach(println)` to show the result after calling `collect` or other action operations; otherwise you'll get unexpected results (like unsorted results).

Result

```
scala> :load /media/jack/File/hadoopHw/hw8/jack/pageRank.scala
Loading /media/jack/File/hadoopHw/hw8/jack/pageRank.scala...
import scala.io.Source
import org.apache.spark.graphx.Graph
import scala.util.hashing.MurmurHash3
defined object PageRank
(1.024373490355488,A)
(1.024373490355488,B)
(0.8016717537932411,C)
(1.472866894710858,D)
(0.8016717537932411,E)
(1.2832802325323818,F)
(1.2832802325323818,G)
(0.9488863714440244,H)
(0.8446560052929347,I)
(0.5149397751899604,J)
```