CSS 534
Program 4: Parallelizing Shortest-Path Search, using a BFS approach on Spark

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose
This programming assignment gets you assimilated to Spark data transformations and actions by parallelizing BFS (breadth-first search)-based shortest-path search.

2. Spark Installation and Lab 4A/B
You need to install Spark independently in your UW1-320 Linux account. Follow the two CSS534 slide sets, (i.e., DataParallel1.ppt and DataParallel2.ppt). You can find spark-2.3.1-bin-hadoop2.7.tgz under the /CSSDIV/classes/534/spark-2.3.1-bin-hadoop2.7 directory. So, you may just copy it to your account rather than download the file from http://spark.apache.org/.

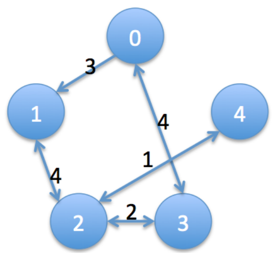
(1) **Lab 4A:** play with pySpark a little, as referring to DataParallel1.ppt's page 6. Thereafter, simply copy /CSSDIV/classes/534/programming/Spark/MyClass.java to your account, compile it, and run it on a single node.

(2) **Lab 4B:** set up a Spark standalone cluster in your account, as referring to DataParallel2.ppt's page 2.
Thereafter, copy /CSSDIV/classes/534/programming/Spark/JavaWordCountFlatMp.java to your account, replace
```
lines.flatMap( s -> Arrays.asList( s.split( " " ) ).iterator() );
```
with
```
lines.flatMap( new FlatMapFunction<String, String>() { … });
```
Then, compile and run it on both a single machine and over parallel machines.

3. BFS-based Shortest-Path Search and Graph Definition
Although Dijkstra's shortest-path search is well known, due to tis dynamic programming aspect, it is quite difficult to parallelize. As discussed in MapReduce, breadth-first search over a given network is easy to parallelize. We will use the same approach but introduce a little more complication to it by giving each graph edge a different weight. For this purpose, we need to define in an input file not only vertex-to-vertex edge connectivity but also its weight. The input file format we would like to use is:

vertexId1=neighbor1,linkWeight1;nieghbor2,linkWeight2;neighbor3,linkWeight3;…
vertexId2=…

For instance, the following 5-vertex graph can be represented in an input file below:



```
0=3,4;1,3
1=0,3;2,4
2=1,4;3,2;4,1
3=0,4;2,2
4=2,1
```

For your convenience, you can use GraphGen.class to generate a random network. Copy GraphGen.class and Map.class into your account and type:
```
java GraphGen #vertices filename
```

Where #vertices is the number of vertices of a graph you want to create and filename is the name of a file that includes all the connectivity and weight information. For example, if you want to create a graph of 3000 vertices and save its information into graph.txt, type:

```
java GraphGen 3000 graph.txt
```

4. Algorithm

The main(String[] args) function receives three arguments: (1) an input file name, (2) source vertex Id, and (3) destination vertex Id. Thereafter read the input file and start a timer for performance measurement.

```
    // start Sparks and read a given input file
    String inputFile = args[0];
    SparkConf conf = new SparkConf( ).setAppName( "BFS-based Shortest Path Search" );
    JavaSparkContext jsc = new JavaSparkContext( conf );
    JavaRDD<String> lines = jsc.textFile( inputFile );

    // now start a timer
    long startTime = System.currentTimeMillis();
```

After reading a graph definition into the `lines` RDD, create `JavaPairRDD<String, Data> network` by calling a transformation, `lines.mapToPair( line -> { … } );` where `String` is each vertex Id such as “0”, “1”, “2”, … and `Data` is a Data instance that includes the corresponding vertex’s attributes as defined below:

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import scala.Tuple2;
/**
 * Vertex Attributes
 */
public class Data implements Serializable {
    List<Tuple2<String,Integer>> neighbors; // <neighbor0, weight0>, ...
    String status;                          // "INACTIVE" or "ACTIVE"
    Integer distance;            // the distance so far from source to this vertex
    Integer prev;                // the distance calculated in the previous iteration

    public Data(){
        neighbors = new ArrayList<>();
        status = "INACTIVE";
        distance = 0;
    }

    public Data( List<Tuple2<String,Integer>> neighbors,
                 Integer dist, Integer prev, String status ){
        if ( neighbors != null ) {
            this.neighbors = new ArrayList<>( neighbors );
        } else {
            this.neighbors = new ArrayList<>( );
        }
        this.distance = dist;
        this.prev = prev;
        this.status = status;
    }
}
```

Note that, when you instantiate each vertex with a Data instance, you have to make sure that the source vertex’s `Data.status` must be “`ACTIVE`”, while all the other vertices’ status should be “`INACTIVE`”. This way your program can execute at least the very first iteration of the following `while` loop:

```
while (t here are any "ACTIVE" vertices ) {
    JavaPairRDD<String, Data> propagatedNetwork = network.flatMapToPair( vertex-> {
        // If a vertex is "ACTIVE", create Tuple2( neighbor, new Data( … ) ) for
        // each neighbor where Data should include a new distance to this neighbor.
        // Add each Tuple2 to a list. Don't forget this vertex itself back to the
        // list. Return all the list items.
    } );

    network = propagatedNetwork.reduceByKey( ( k1, k2 ) ->{
        // For each key, (i.e., each vertex), find the shortest distance and
        // update this vertex' Data attribute.
    } );

    network = network.mapValues( value -> {
        // If a vertex' new distance is shorter than prev, activate this vertex
        // status and replace prev with the new distance.
   } );
}
```

The above **`while`** loop includes three RDD Transformations: (1) flatMapToPair to create a network that received all distance propagations from the current vertices to their neighbors, (2) reduceByKey to reduce all redundant, identical keys, (i.e., vertices) to a single vertex, and (3) mapValues to activate only vertices whose distance got shorter than prev.

After completing this **`while`** loop, stop the timer to measure the time elapsed for the program execution. Don't forget to retrieve the destination vertex' distance. That is the shortest path to find.

5. Coding, Compilation, Input Set-up, and Execution

(1) Coding: save your Java source as ShortestPath.java.

(2) Compilation: you may use Maven or directly compile as:

```
javac -cp jars/spark-core_2.11-2.3.1.jar:jars/spark-sql_2.11-
2.3.1.jar:jars/scala-library-2.11.8.jar:google-collections-1.0.jar:.
ShortestPath.java

jar -cvf ShortestPath.jar ShortestPath.class Data.class
```

(3) Execution:

```
java GraphGen 3000 graph.txt

Spark-submit --class ShortestPath --master spark://uw1-320-01:50763 --executor-
memory 1G --total-executor-cores 1 ShortestPath.jar graph.txt 0 2999
```

(4) Output:

Sample outputs are:

```
Spark-submit --class ShortestPath --master spark://uw1-320-01:50763 --executor-
memory 1G --total-executor-cores 1 ShortestPath.jar graph.txt 0 1500

from 0 to 1500 takes distance = 41

Spark-submit --class ShortestPath --master spark://uw1-320-01:50763 --executor-
memory 1G --total-executor-cores 1 ShortestPath.jar graph.txt 0 2999

from 0 to 2999 takes distance = 32
```

Note that the executor memory of 1G is enough for a network of 3000 vertices. But if you want to test your program with a larger network, you might want to increase the size.

6. Statement of Work

Step 1: Code your ShortestPath.java. You may use: GraphGen.java, Map.java, and Data.java. For your initial implementation, follow the algorithm shown above and go to Step 2. If your time allows, you may improve the execution performance of your program by modifying the above code framework or even using a different approach you would like to try.

Step 2: Compare the performance of the following 12 different executions of your ShortestPath program, using a graph of 3000 vertices. The following table shows the performance of the professor's key answer execution. The best performance improvement by parallel execution is **2.233 times** faster with 2 computing nodes, using 4 cores in total than a sequential execution.

# computing nodes	Ave # cores per node	Total # cores	Elapsed time (sec)	Performance improvement
1	1	1	441.046	1.000
1	2	2	443.556	0.994
1	4	4	440.757	1.001
1	8	8	451.728	0.976
2	1	2	330.257	1.335
2	**2**	**4**	**197.475**	**2.233**
2	4	8	209.076	2.110
2	8	16	230.990	1.909
4	1	4	327.908	1.345
4	2	8	237.429	1.858
4	4	16	209.748	2.103
4	8	32	234.909	1.878

Step 3: Discuss about the following items in your report:

(1) Mandatory discussion on pros and cons of the original algorithm/code framework shown in this homework specification.
  a. Discussion from the programmability viewpoint.
  b. Discussion from the viewpoint of execution performance.

(2) Optional discussion on pros and cons of your improved algorithm if you implemented any.
  a. Discussion from the programmability viewpoint.
  b. Discussion from the viewpoint of execution performance.

7. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	**Grade**
Documentation of your BFS-based shortest-path search program in just **one page please**.	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. • 25pts: well-organized and correct code receives • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs or incomplete code receives	25pts
Execution output that verifies the correctness of your implementation and demonstrates any	25pts

improvement of your program's execution performance. • 25pts: Correct execution and better performance than your professor's program execution (2.23). It should be at least 5% better in terms of performance improvement, (i.e., 2.34 or better). • 23pts: Correct execution and the same performance as your professor's program execution (between -5% and +5%, i.e., between 2.11 and 2.34). • 20pts: Correct execution but the slower performance than your professor's program execution, (i.e., -5% or slower = 2.11 or worse). • 15pts: Wrong or incomplete work.	
Discussions about in one page. • Pros and cons of program 4's original algorithm o Programmability (10pts) o Execution performance (10pts) • Optional: pros and cons of your improved algorithm if you implemented any. o Programmability (extra 2pts) o Execution performance (extra 2pts)	20pts
Lab Session 4A Please turn in your lab 4A and 4B together with date of program 4. Your source code and execution outputs are required. • 4A submitted (5pts) • 4B submitted (5pts)	10pts
Total Note that program 4 takes 15% of your final grade.	100pts