Programming 4 Assignment Report

Student Name: Dazhi Li

Student NETID: dazhili

Student ID#: 2400330

# 1. Documentation

This part briefly describes how I implement my Shortest Path program by Spark.

All programs were run on cssmpi8h(master)~cssmpi12h

The most important part to implement the BFS algorithm on Spark is doing correct RDD manipulations. The standard RDD receives text from graph.txt as the input of the whole program. Then I transformed the RDD to PairRDD to do the network propagation. The whole RDD transformation is shown below as in fig1.

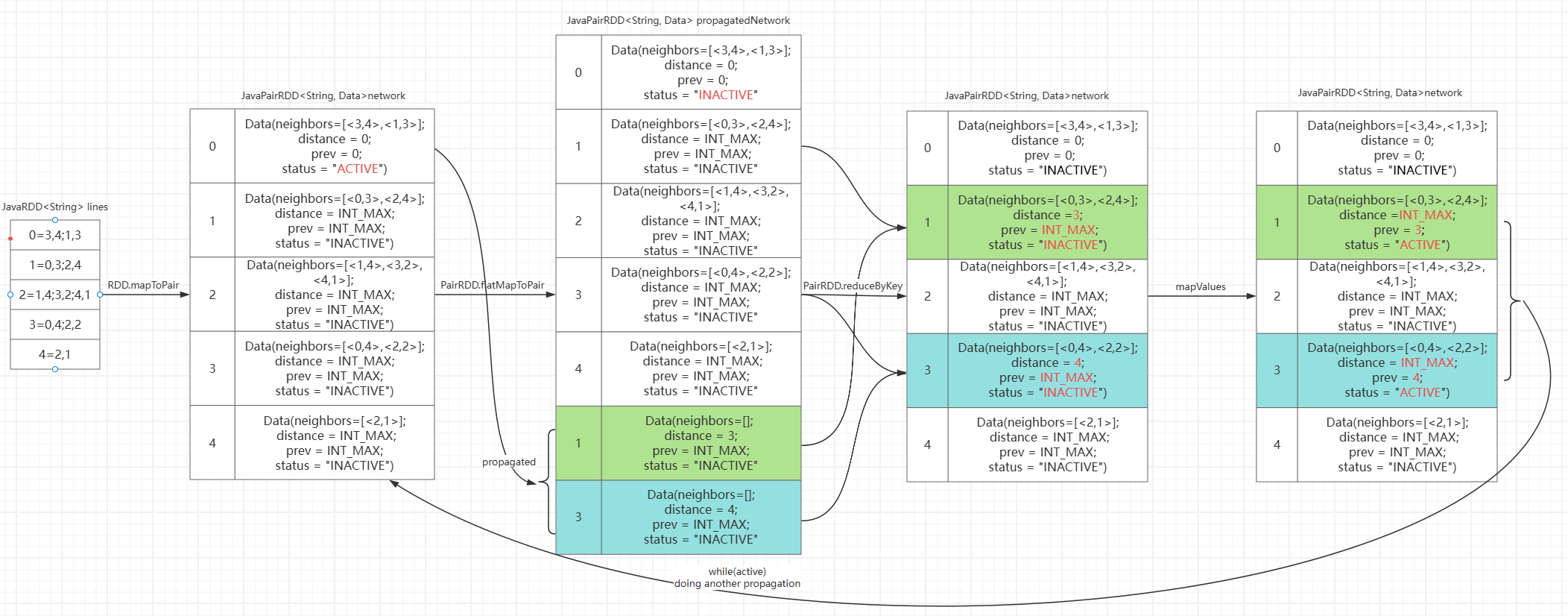


Fig1. RDD transformation examples

Fig1. shows an example of what is happening between each RDD. If 0 is the source vertex, it propagated two neighbors in the network. Then reduce it by the same key to update vertex’s distance. Then using mapValue() to update the value if distance is less than previous distance. Also, modify the status to active. While active nodes in the network, the program keep propagating new neighbors and going on.

The whole program stops when there is no active node in the network. Then I did go through the RDD to find the destination node and its previous distance as my answer for the shortest path from the source node to the destination node.

# 2.Source Code

Please note, I modified Data.java (adding some getters and setters) and recompiled it for my convenience.

All resource codes could be accessed in the zip file.

**ShortestPath.java**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Iterator;

import java.util.List;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.PairFunction;

import org.apache.spark.api.java.function.PairFlatMapFunction;

import org.apache.spark.api.java.function.Function2;

import org.apache.spark.api.java.function.Function;

import scala.Tuple2;

public class ShortestPath {

    public static boolean is\_still\_active(JavaPairRDD<String, Data> network) {

        return network.filter(d -> d.\_2().is\_active()).count() > 0 ;

    }

    public static void main(String[] args) {

        // start Sparks and read a given input file

        String inputFile = args[0];

        String srcVtx = args[1];

        String dstVtx = args[2];

        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();

        // initializing the whole grogram: loading data

        JavaPairRDD<String, Data> network =  lines.mapToPair(

            new PairFunction<String, String, Data>() {

                @Override

                public Tuple2<String, Data> call(String line) {

                    // splitOne: ["0", "3,4;1,3"] -> ["source vertext", "neighbors info"]

                    String[] splitOne = line.split("=");

                    String vtx = splitOne[0];

                    List<Tuple2<String, Integer>> neighbors = null;

                    if (splitOne[1] != null && splitOne[1].length() != 0) {

                        neighbors = new ArrayList<Tuple2<String, Integer>>();

                        // singleEdge: "3,4" -> "nextVertex, nextVertexEdgeLength"

                        for (String singleEdge : splitOne[1].split(";")) {

                            // splitTwo: ["3", "4"] -> ["nextVertex", "nextVertexEdgeLength"]

                            String[] splitTwo = singleEdge.split(",");

                            neighbors.add(new Tuple2<>(splitTwo[0], Integer.valueOf(splitTwo[1])));

                        }

                    }

                    // specialize the source index to initiate the program

                    Data singleNetwork = null;

                    if (srcVtx.equals(vtx)) {

                        singleNetwork = new Data(neighbors, 0, 0, "ACTIVE");

                    } else {

                        singleNetwork = new Data(neighbors, Integer.MAX\_VALUE, Integer.MAX\_VALUE, "INACTIVE");

                    }

                    return new Tuple2<>(vtx, singleNetwork);

                }

            }

        );

        while (is\_still\_active(network)) {

            JavaPairRDD<String, Data> propagatedNetwork = network.flatMapToPair(

            // 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.

            new PairFlatMapFunction<Tuple2<String, Data>, String, Data>() {

                @Override

                public Iterator<Tuple2<String, Data>> call(Tuple2<String, Data> vertex) {

                    List<Tuple2<String, Data>> results = new ArrayList<>();

                    // Doing propagation

                    if (vertex.\_2().is\_active()) {

                        for (Tuple2<String,Integer> neighbor : vertex.\_2.getNeighbors()) {

                            results.add(new Tuple2<>(neighbor.\_1(), new Data(null, vertex.\_2().getPrev() + neighbor.\_2(), Integer.MAX\_VALUE, "INACTIVE")));

                        }

                        vertex.\_2().setStatus("INACTIVE");

                        results.add(vertex);

                    } else {

                        results.add(vertex);

                    }

                    return results.iterator();

                }

            }

            );

            network = propagatedNetwork.reduceByKey(

            // For each key, (i.e., each vertex), find the shortest distance and

            // update this vertex’ Data attribute.

            new Function2<Data, Data, Data>() {

                @Override

                public Data call(Data k1, Data k2) {

                    Integer newDistance = 0;

                    Integer newPrev = 0;

                    List<Tuple2<String,Integer>> newNeighbors = null;

                    // calculating new distance

                    if (k1.getDistance() < k2.getDistance()) {

                        newDistance = k1.getDistance();

                    } else {

                        newDistance = k2.getDistance();

                    }

                    // remain existed prev

                    if (k1.getPrev() != Integer.MAX\_VALUE) {

                        newPrev = k1.getPrev();

                    } else {

                        newPrev = k2.getPrev();

                    }

                    // remain existed neighbors

                    if (k1.getNeighbors().isEmpty()) {

                        newNeighbors = k2.getNeighbors();

                    } else {

                        newNeighbors = k1.getNeighbors();

                    }

                    return new Data(newNeighbors, newDistance, newPrev, "INACTIVE");

                }

            }

            );

            network = network.mapValues(

            // If a vertex’ new distance is shorter than prev, activate this vertex

            // status and replace prev with the new distance.

            new Function<Data, Data>() {

                @Override

                public Data call(Data value) {

                    if (value.getDistance() < value.getPrev()) {

                        value.setPrev(value.getDistance());

                        value.setStatus("ACTIVE");

                    }

                    value.setDistance(Integer.MAX\_VALUE);

                    return value;

                }

            }

            );

        }

        // ending time

        long endTime = System.currentTimeMillis();

        // Print out results

        System.out.println("Time Elapsed = " + (endTime - startTime));

        List<Tuple2<String, Data>> finalNetwork = network.collect();

        for (Tuple2<String, Data> tuple: finalNetwork) {

            if (tuple.\_1().equals(dstVtx)) {

                System.out.println("from " + srcVtx + " to " + dstVtx + " takes distance = " + tuple.\_2().getPrev());

            }

        }

    }

}

**Data.java**

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>, <neighbor1, weight1>, ...

    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;

    }

    public boolean is\_active() {

        return "ACTIVE".equals(status);

    }

    public List<Tuple2<String,Integer>> getNeighbors() {

        return neighbors;

    }

    public Integer getPrev() {

        return prev;

    }

    public void setPrev(Integer prev) {

        this.prev = prev;

    }

    public Integer getDistance() {

        return distance;

    }

    public void setDistance(Integer distance) {

        this.distance = distance;

    }

    public void setStatus(String status) {

        this.status = status;

    }

}

# 3.Execution Result

## 3.1 Correct Execution

As I have compared professor’s graph.txt and re-run GraphGen.java multiple times, I am sure that GraphGen generates the same results when number of vertices are same. As a result, my graph.txt is the same as what professor’s. So, my Shortest Path result should be the same as the professor’s, as shown in Fig2.

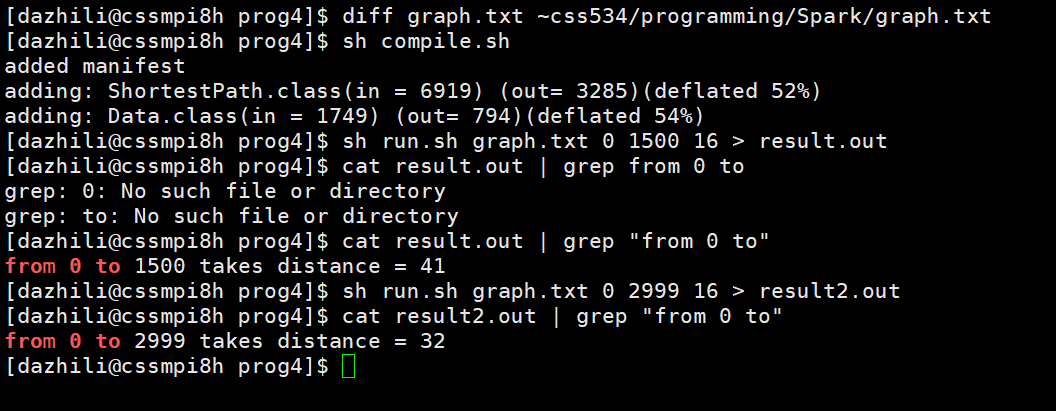


Fig2. Shortest path result on 0 to 1500 and 0 to 2999

## 3.2 Performance evaluation

I did evaluation from 1 computing node to 5 computing nodes with 1 average cores to 4 average cores respectively. The execution results are shown in table below. Please note, the time is measured by java System.currentTimeMillis() not by the job running result in Spark. And the graph is the 3000 vertices graph with a source node from 0 to destination node of 1500.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # computing nodes | Ave # cores per node | Total # cores | Elapsed time (sec) | Performance improvement |
| 1 | 1 | 1 | 210.730 | 1 |
| 1 | 2 | 2 | 153.439 | 1.373 |
| 1 | 3 | 3 | 153.765 | 1.370 |
| 1 | 4 | 4 | 151.701 | 1.389 |
| 2 | 1 | 2 | 124.871 | 1.688 |
| 2 | 2 | 4 | 116.299 | 1.811 |
| 2 | 3 | 6 | 127.690 | 1.650 |
| 2 | 4 | 8 | 117.800 | 1.789 |
| 3 | 1 | 3 | 119.566 | 1.762 |
| 3 | 2 | 6 | 115.262 | 1.828 |
| 3 | 3 | 9 | 117.144 | 1.798 |
| 3 | 4 | 12 | 120.677 | 1.746 |
| 4 | 1 | 4 | 123.293 | 1.709 |
| 4 | 2 | 8 | 126.486 | 1.666 |
| 4 | 3 | 12 | 123.050 | 1.713 |
| 4 | 4 | 16 | 123.183 | 1.711 |
| 5 | 1 | 5 | 121.501 | 1.734 |
| 5 | 2 | 10 | 123.800 | 1.702 |
| 5 | 3 | 15 | 120.729 | 1.745 |
| 5 | 4 | 20 | 121.485 | 1.735 |

The slowest result as shown in fig3:

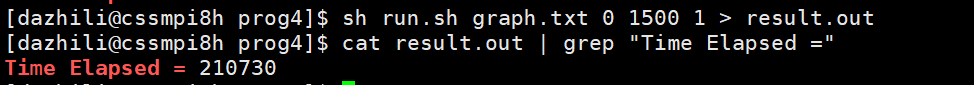


Fig3. 1 node 1 core running result

The fastest result as shown in fig4:

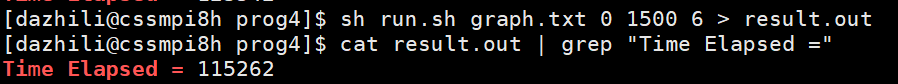


Fig4. 3 node 6 cores running result (avg 2 cores per node)

The performance improvement is 1.828

# 4.Discussions

Discussion on pros and cons of the original algorithm/code framework shown in this homework specification.

## 4.1 Programmability

**Pros**: Using lambda functions in spark makes it easier for programmer to write a program and understand how function works in a RDD transformation. Also, the original Data.java provides a basic framework on data processing.

**Cons**: Using tuple2 key value pair like tuple.\_1() and tuple.\_2() increases the difficulty for reader to understand the code without annotations. Moreover, using class methods of JavaRDD and JavaPairRDD requires user to search online for instructions of those methods. This information includes variable types, input and output that a method accepts.

## 4.2 Execution performance

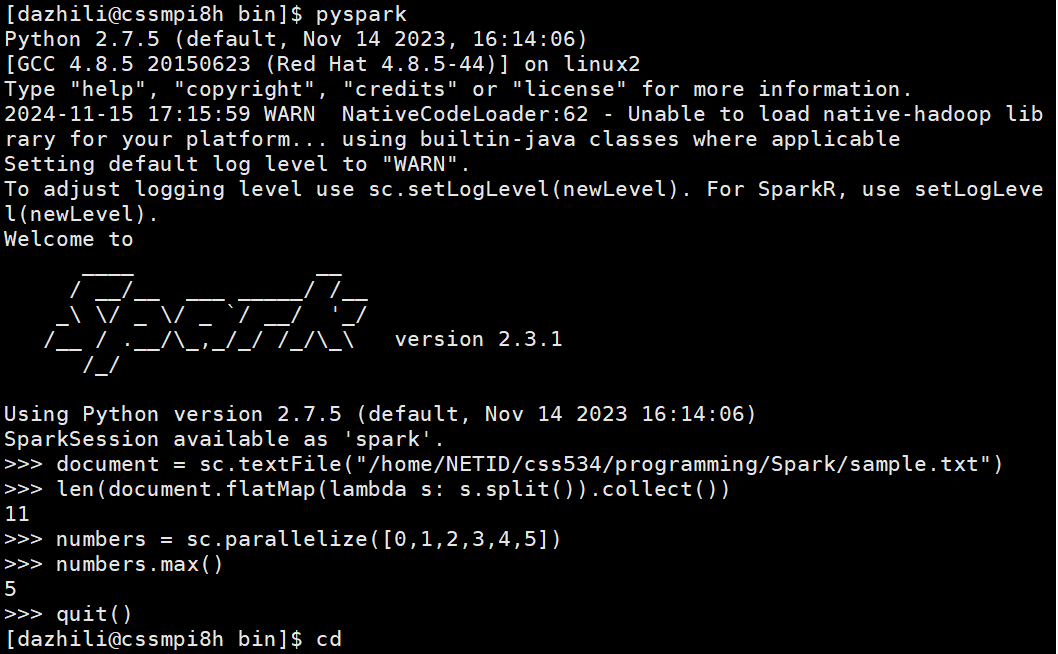
**Pros**: As we have discussed in the class, replacing lambda functions with anonymous inner class is the most efficient way to improve the performance. Anonymous inner class will separate methods to each node before runtime while lambda function serializes it while runtime.

**Cons**: There is no accumulator/broadcast variables in my current code. This requires imposes more communications between nodes. As a result, decrease the performance of the code. Moreover, using Data.java in every step is a waste of memory. For an example, when propagating networks, my program generates new JavaRDDPair with a whole Data class in it. However, what really matters is only one string which indicates the next node and one Integer which indicates the distance so far. This redundant data structure will absolutely reduce the programming performance.

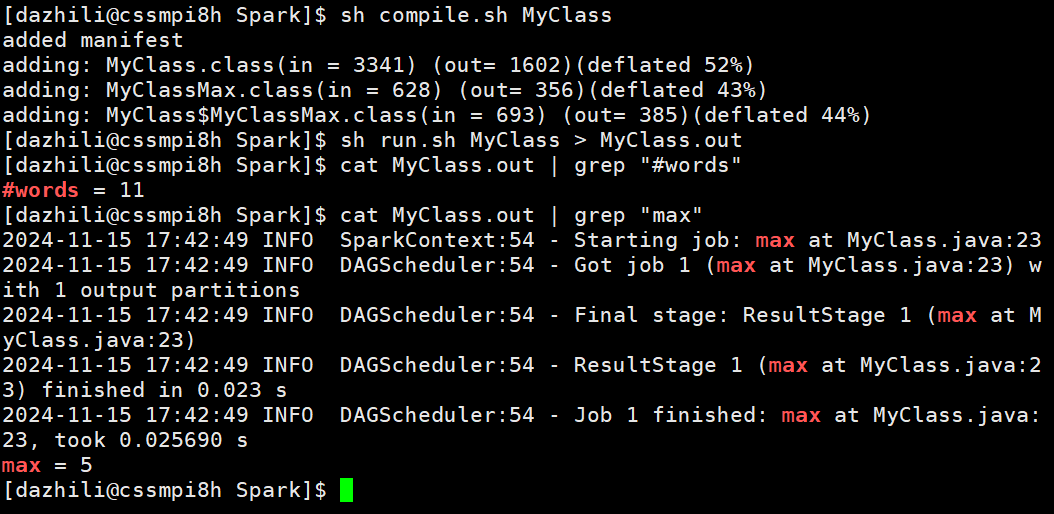
# 5.LAB4 Details

## 5.1 LAB4A

Play with pySpark:

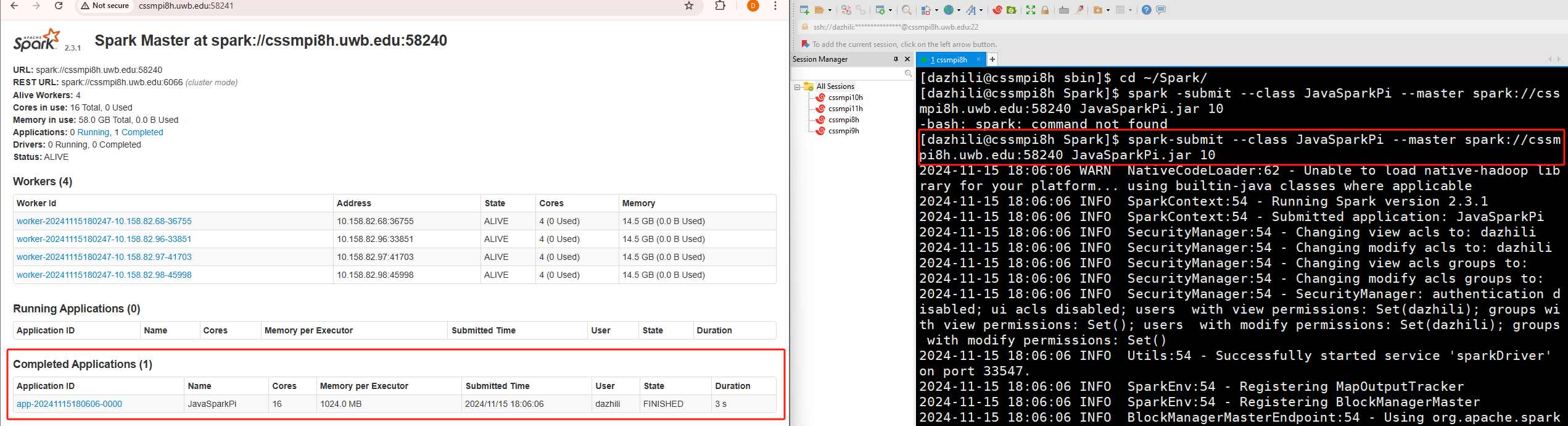


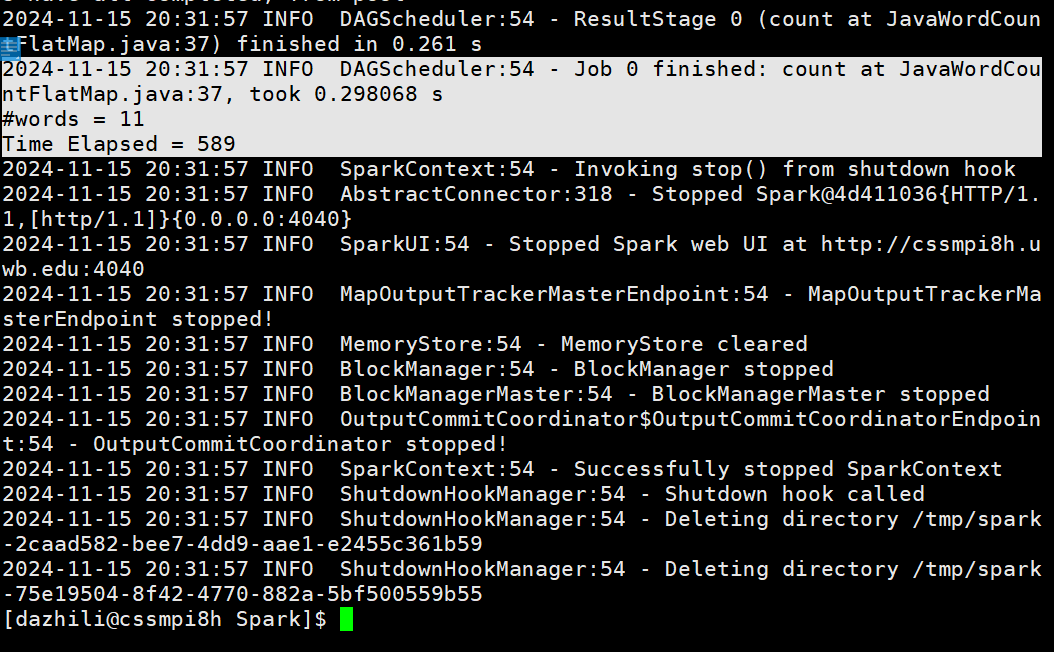
1 node:



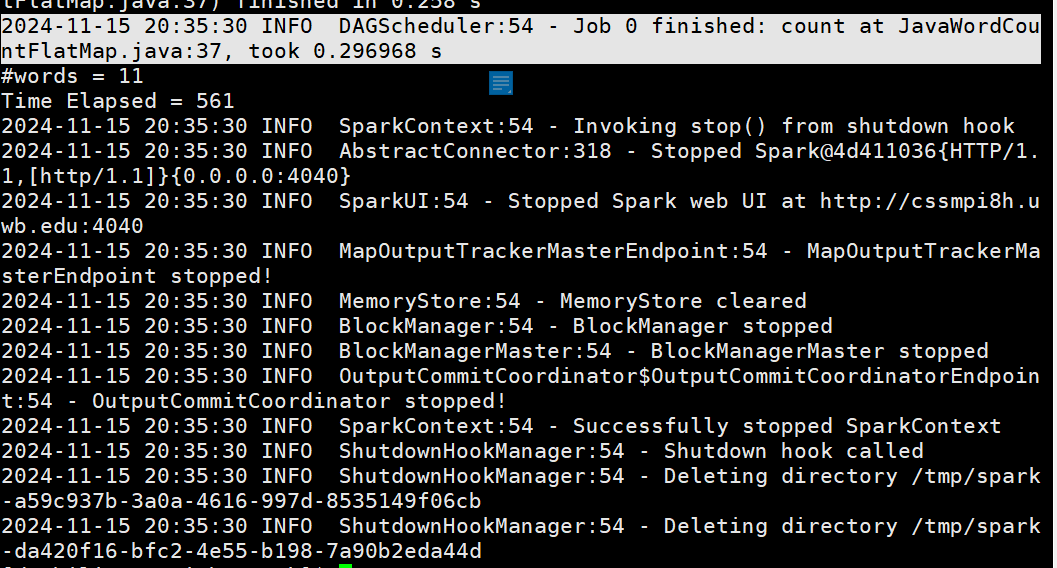
## 5.2 LAB4B

4 nodes:





1 node:



Lab4B shows performance improvement by 4 nodes in spark’s own running time. However, on my self-set timer, it is slower than 1 node. This result could attribute to the small datasets and different performance conditions on those virtual machines.

Source code:

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import java.util.Arrays;

import org.apache.spark.api.java.function.FlatMapFunction;

import java.util.Iterator;

public class JavaWordCountFlatMap {

    public static void main(String[] args) {

        // configure spark

        SparkConf sparkConf = new SparkConf().setAppName("Java Word Count FlatMap")

        .setMaster("local[2]").set("spark.executor.memory","2g");

        // start a spark context

        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        // provide path to input text file

        String path = "sample.txt";

        // start time

        long startTime = System.currentTimeMillis();

        // read text file to RDD

        JavaRDD<String> lines = sc.textFile(path);

    // Java 8 with lambdas: split the input string into words

    JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {

        @Override public Iterator<String> call(String s)

        {

            return Arrays.asList(s.split(" ")).iterator();

        }

    });

    // stop time

    long stopTime = System.currentTimeMillis();

    // print #words

    System.out.println( "#words = " + words.count( ) );

    // print time

    System.out.println("Time Elapsed = " + (stopTime - startTime));

    }

}

# 6.Summary

In this report I briefly documented the shortest path program and how I implement it with Spark. I discuss on how my current code influences on programmability and execution performance in multiple dimensions. Moreover, I evaluated how spark performance varies on slaves’ number and execution cores numbers and got an optimal performance improvement on three workers with total 6 cores. Finally, I posted my lab4 results and discussed about how some results were formed.