Justin Chen
CS549 - A
I pledge my honor that I have abided by the Stevens Honor System.

Hadoop EMR Report

I wasn't able to get the Virtual Machine running locally on my computer, so I did all my testing (for both the small test graph and the Wikipedia PageRank) on an AWS EMR instance that had Hadoop pre-installed and pre-configured.

For the exact commands, I attached a video podcast detailing how I set up the EMR.

Mappers and Reducers

This is what I did for each of my mappers and reducers. We mostly just followed the specifications - especially the part that gave hints on the inputs and outputs for the mappers and reducers:

InitMapper

Input: [vertex]: [space delimited adjacency list]

Output: [vertex], [space delimited adjacency list]

```
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException,
        IllegalArgumentException {
    String line = value.toString(); // Converts Line to a String
    /*
     * TODO: Just echo the input, since it is already in adjacency list format.
     * Alternatively, output adjacency pairs that will be collected by reducer.
     */

    // Input is NodeID and space delimited adjacency list split by :
    // We probably want to trim the second list in case space delimeters go wrong

    String[] divider = line.split( regex: ":");
    context.write(new Text(divider[0].trim()), new Text(divider[1].trim()));

}
```

InitReducer

Input Key: Vertex

Input Value: Array of adjacency lists

Output Key: [vertex];[rank]

Output value: Space delimited list of vertices involved (adjacent)

```java
public void reduce(Text key, Iterable<Text> values, Context context) throws IO
    /*
     * TODO: Output key: node+rank, value: adjacency list
     */

    // Input Key is the vertex
    // The input value is an array of adjacency lists

    // Output key is vertex;rank (1 to begin with)
    // Output value is the space seperated list of vertices
    // The hint wanted this to be comma delimited
    // but I didn't want to mess with it since it's already space delimited an

    for (Text adjacencyList : values)
    {
        context.write(new Text((key + ";1").trim()), adjacencyList);
    }
}
```

IterMapper

Input Key: [NodeID];[Rank]

Input Value: Space delimited adjacency list

There can be two possible outputs:

The first one

Output Key: Node ID

Output Value: @[Space delimited adjacency list]

The second one

Output Key: A single adjacent vertex

Output Value: Computed weights

```
// Input key: NodeID;rank
// Input value: space delimited adjacency list

// Output 1 key: Adjacent vertex (item in adjacency vertex)
// Output 1 value: Computed weights
// 🚨 I tried outputting normal weights and then calculating the computed weights in the IterReducer
// 🚨 But the math broke and I don't know why so we're doing it in IterMapper

// Output 2 key: NodeID
// Output 2 value: @ + Space delimited adjacency list (instead of stuff in back)

String[] nodeSplit = sections[0].split( regex: ";");
String nodeID = nodeSplit[0];
String nodeWeight = nodeSplit[1];

String adjacencyListStr = sections[1];
String[] adjacencyList = sections[1].split( regex: " ");

// Calculate equal weight split
// The below is suggested by Intellij intellisense
double weight = Double.parseDouble(nodeWeight);
double equalWeight = weight / adjacencyList.length;

// For all adjacent vertex, write the equal weight
for (String adjacentVertex : adjacencyList)
{
    context.write(new Text(adjacentVertex), new Text( string: "" + equalWeight));
}

// Write NodeID and @ + space delimited adjacency list
context.write(new Text(nodeID), new Text( string: "@" + adjacencyListStr));
```

IterReducer

Input Key: Vertex ID

The input value is an array. However, there's two possible input values inside these arrays.

The first one

Input Value: Adjacent Vertices

The second one

Input Value: Weights

Output Key: [VertexID];[Rank]

Output Value: Space delimited adjacency list

```java
/*
 * TODO: emit key:node+rank, value: adjacency list
 * Use PageRank algorithm to compute rank from weights contributed by incoming edges.
 * Remember that one of the values will be marked as the adjacency list for the node.
 */
double d = PageRankDriver.DECAY; // Decay factor
double rank = 0.0; // stores the decay factor in a variable rank

// Input Key: Vertex ID
// Input Value: Array with [Adjacent Vertices (w/ equal weights) and space-seperated adjacency lists]

// Output Key: VertexID;Rank (Compute Rank from weights)
// Output Value: Space-Seperated adjacency list

// Declare some variables we'll process later
String nodeID = key.toString();
String adjacencyList = "";

for (Text inputVal : values)
{
    String valueStr = inputVal.toString();
    if (valueStr.startsWith("@"))
    {
        // Here we have space-seperated adjacency list
        // Remove le delimiter here
        adjacencyList = valueStr.replaceAll( regex: "@", replacement: "");
    }
    else
    {
        // Here we have equal weights of adj vertices
        rank += Double.parseDouble(valueStr);
    }
}
```

DiffMap1

The input and outputs are neatly explained in the comments.

```java
/**
 *  TODO: read node-rank pair and emit: key:node, value:rank
 */

// Input Key: Line Number (Ignored)
// Input Value 1: vertex;rank
// Input Value 2: Space delimited list of adjacent vertices

// Output Key: Node
// Output Value: Rank

// I think we just parse input value 1 and ignore input value 2
String[] delinParse = sections[0].split( regex: ";");
context.write(new Text(delinParse[0]), new Text(delinParse[1]));
```

DiffRed1

Input Key: Vertex ID

Input Value: List with exactly two rank values for avertex to calculate difference from

Output Key: Vertex ID

Output value: Difference between ranks

We slightly deviated from the documentation because I was running into errors in AWS EMR. I asked a friend and they suggested that I just keep the difference in the value because that's how it gets treated in DiffMap2.

```
    // Input Key: Vertex
    // Input Value: List with two rank values for a vertex

    // Output Key: Difference between ranks
    // Output Value: Reduce

    // Chuck the values into a ranks array
    int i = 0;
    for (Text value : values)
    {
        if (i < 2)
        {
            ranks[i] = Double.parseDouble(value.toString());
            i += 1;
        }
        else
        {
            throw new IOException("Input size of values for DiffRed1 must be 2.");
        }
    }

    // Calculate the absolute value of rank differences
    double absDiff = Math.abs(ranks[0] - ranks[1]);

    // TODO: Output
    context.write(key, new Text(Double.toString(absDiff)));
}
```

DiffMap2

The inputs and outputs should be explained in the comments.

```
/*
 * TODO: emit: key:"Difference" value:difference calculated in DiffRed1
 */

// Input key: Line # (no one cares)
// Input value: Rank difference

// Output key: "Difference"
// Input key: Difference value as text

context.write(new Text( string: "Difference"), new Text(sections[1]));
```

DiffRed2

Input Key: “Difference”

Input value: List of rank differences for each vertex

Output Key: Null

Output Value: Maximum difference between the ranks

Again, I tried following the specs but my EMR instance kept crashing. I used the same trick from earlier since I didn’t want to revamp the entire code to match the specifications document (and also because it would cost a lot of money on AWS EMR to debug the code)

```
// Input key: "Difference"
// Input value: List of rank differences for each vertex

// Output key: Maximum difference between the values
// Output value: null

for (Text val : values)
{
    double diff = Double.parseDouble(val.toString());

    if (diff > diff_max)
    {
        diff_max = diff;
    }
}

// TODO: Output
// context.write(new Text(diff_max + ""), null);
context.write(new Text(), new Text( string: diff_max + ""));
```

JoinNameMapper

This was already provided by Professor Duggan. I didn't change this.

JoinRankMapper

This was already provided by Professor Duggan. I didn't change this.

JoinReducer

Input Key: Vertex ID

Input Value: List with vertex names and vertex rank (name, then rank)

Output Key: Vertex Name

Output Value: Vertex Rank

```
// TODO values should have the vertex name and the page rank (in that order).
// Emit (vertex name, pagerank) or (vertex id, vertex name, pagerank)
// We're going to emit (vertex name, pagerank)
// Ignore if the values do not include both vertex name and page rank

// Input Key: Vertex ID
// Input Value: List with vertex name and vertex rank

// Output Key: Vertex Name
// Output Key: Vertex Rank

String vertexName = "";
String vertexRank = "";

// We don't know why values is an iterable
// Ask Duggan what to do here

// For now, we're treating this as a normal iterable but reduce only sends one output
int i = 0;
for (Text value : values)
{
    if (i == 0)
    {
        vertexName = value.toString();
    }
    else if (i == 1)
    {
        vertexRank = value.toString();
    }

    i++;
}

if (vertexName != null && vertexRank != null)
{
    context.write(new Text(vertexName), new Text(vertexRank));
}
```

FinMapper

The input and output for FinMapper is in the comments.

However, I do want to just point out that we temporarily set the rank to a negative number in order to sort it by descending order.

```
/*
 * TODO output key:-rank, value: node
 * See IterMapper for hints on parsing the output of IterReducer.
 */

// Input Key: Line Number
// Input Value: Vertex;Rank and space seperated list of adjacent vertices

// output key: -rank (to sort in reverse order)
// Output vertex: Vertex

if (sections.length > 2)
{
    throw new IOException("Incorrect data format");
}
if (sections.length != 2) {
    return;
}

context.write(new DoubleWritable( value: -1 * Double.parseDouble(sections[1])), new Text(sections[0]));
```

FinReducer

Input and Output should both be self-explanatory.

But we revert the negative PageRank number back to the positive number.

key.get() was suggested by Intellij Intellisense so I stuck with it.

```
public void reduce(DoubleWritable key, Iterable<Text> values, Context context) throws IOException,
        InterruptedException {
    /*
     * TODO: For each value, emit: key:value, value:-rank
     */

    // Input key: Negative Page Rank (-rank)
    // Input value: List of vertices with that page rank

    // Output Key: Vertex
    // Output Value: -rank for each vertex in the list

    for (Text value : values)
    {
        context.write(new Text(value.toString()), new Text( string: "" + (-1 * key.get())));
    }
}
```

MODIFIED CODE FROM PageRankDriver

An important thing to note: I modified `PageRankDriver.java` because I was getting errors when my program kept trying to parse names like `Node_1` into a long.

This mainly happened in the `Summarize_Result()` function.

It turns out that this code was written to handle <Vertex ID, PageRank> - both of which are numerical values. If we wanted this to handle names like `Node_1`, we must modify it to handle Strings.

So I changed the HashMap type and got rid of parseLong (or something similar to that function).

```
HashMap<String, Double> values = new HashMap<String, Double>();
// HashMap to store Node - Rank Pairs
```

I also added a new CLI command for the `.jar` file to debug my new changes:

```
} else if (args.length == 2)
{   // Justin debugging script to summarize
    if (job.equals("summarize")) {
        summarizeResult(args[1]);
    }
}
```

Testing the Code

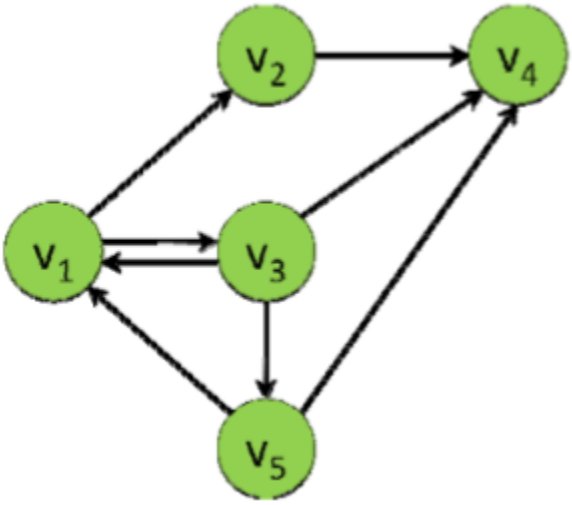
As I mentioned earlier, I wasted money on AWS EMR since I couldn't run Hadoop locally. Professor Duggan mentioned this was an acceptable alternative if I can't download the Virtual Machine (although it did cost a lot more money).

I wasn't able to directly add print statements or trace the .jar file since EMR handled all of that (and I don't know if it's possible to see `stdout` for what the other Hadoop EC2 instance is doing), so I just threw `IOExceptions` in place of debug statements.

I first tested everything on the small graph to save time and money before moving on to the giant Wikipedia.

Testing the Small Graph

For this graph:



I created an adjacency list:

```
1: 2 3
2: 4
3: 1 4 5
5: 1 4
```

And I created another file for the names:

```
1: Node_1
2: Node_2
3: Node_3
4: Node_4
5: Node_5
```

I provided a video to show exactly how I set up these files in the Hadoop FS. However here's some general commands:

```
hadoop fs -mkdir /users
hadoop fs -mkdir /users/Hadoop
hadoop fs -mkdir input
hadoop fs -put ___ input
hadoop fs -mkdir names
hadoop fs -put ___ names

hadoop jar pagerank.jar composite input output inter1 inter2 names
diffdir 5
```

Here's the PageRank of the small graph:

```
[hadoop@ip-10-0-0-14 ~]$ hadoop fs -cat output/output.txt
Node_4  1.708333333333333
Node_1  0.8583333333333333
Node_2  0.575
Node_3  0.575
Node_5  0.43333333333333335
```

And here's the running time.
I used 3 reducers.

```
Total time spent by all maps in occupied slots (ms)=1084272
Total time spent by all reduces in occupied slots (ms)=2046144
Total time spent by all map tasks (ms)=22589
Total time spent by all reduce tasks (ms)=21314
```

I don't think these numbers are particularly accurate since it took me around 3-5 minutes to run this MapReduce. However, according to AWS EMR documentation, their Hadoop reducers/mappers run in parallel so that could explain the discrepancy.

Wikipedia Data

I ran the Wikipedia Data through AWS EMR.
You can see the setup (I used `wget` to fetch the data) in an attached video.

I attached the `output.txt` for the wikipedia PageRank in the ZIP file. Here's a screenshot of the results:

```
[hadoop@ip-10-0-0-14 ~]$ hadoop fs -cat output/output.txt
United_States   12689.477593863145
2007    8089.965342465858
2008    7803.829442056091
Geographic_coordinate_system    7192.123464318788
United_Kingdom  5794.897724513176
2006    4973.640453667824
France  4195.769150260461
Wikimedia_Commons       4147.856042431833
Wiktionary      3744.356523396049
Canada  3733.4092199148427
```

I tested this using 10 and 50 reducers.

- When I used 10 reducers, this took more than 8 hours (I left it running overnight as a daemon and it was still running when I woke up)
- When I used 50 reducers, this took around 4 hours

This is probably because AWS EMR runs Hadoop in parallel. When you have more reducers, you effectively "load balance" each reducer by splitting up the massive amounts of data they need to process between more reducers and hence - the MapReduce runs faster.