EE 360P: Concurrent and Distributed Systems **Vijay K. Garg**
Spring 2023 **TA's: Robert Streit , Kavya Varma, Sidharth Nair**

Assignment 4
Deadline: Tuesday, April 11th, 2pm

In this assignment, you will use Apache Spark to analyze a text dataset. Specifically, we will build a directed graph where the vertices are words in the dataset, and edges are directed from a predecessor to a successor in a pair of co-occuring words (i.e. words which are adjacent in the text). Furthermore, each edge will be weighted by the frequency of that co-occurence. On large datasets this can be a costly task, however we will map-reduce based distributed computations to accelerate this.

Your solution will be implemented in `WordGraph.java`. We've provided a template file on the course Github. We are also providing a docker image with Apache Spark, and scripts to run your program can also be found on the course Github. Finally, this assignment should be done in teams of two. The Jar and Java files have to be zipped into a file named `EID1_EID2.zip` and submitted through Canvas.

1 Implementing the Word Graph

Your code will build a table that represents a graph giving the frequncies of words occurring adjacently in a text. The data set for this problem is the novel Pride and Prejudice. A collection of files (one per chapter) is provided in the course Github. The occurrences are calculated in individual lines; your program does not need to intelligently separate sentences because the Mapper function is invoked for each newline by default.

When two words are adjacent to each other in a sentence we will refer to first as the *predecessor* and the second as the *successor*. Moreover, an ordered predecessor-successor pair is a *co-occurrence*. For example, consider the following sentence:

Inconsistent premises always make an argument valid.

In the above sentence we have the co-occurrences (inconsistent, premises), (premises, always), (always, make), (make, an), (an, argument), and (argument, valid) where in each of these tuples I've made the first entry the predecessor and the second the successor.

Now, we wish to construct a directed graph. The vertices will be words in the text, and an edge directed from predecessor to successor will exist if two words occur adjacently in the text. Each edge will be weighted by the frequency that this co-occurrence appears, with respect to all other co-occurrences of the same predecessor. This is to say, you will weight an edge from the predecessor *p-word* to the successor *s-word* with the following formula:

$$weight(p\text{-}word, s\text{-}word) = \frac{count\ of\ co\text{-}occurrences\ (p\text{-}word, s\text{-}word)}{total\ count\ of\ co\text{-}occurrences\ with\ predecessor\ p\text{-}word},$$

where the counts in the above formula are taken with respect to the *entire* input text.

In effect, what we've done is compute the "likelihood" of a successor word being the next word in the text, given a predecessor word. Such a computation is the basis for training artificial intelligence models generating synthetic text.

To make this assignment more straightforward, please do the following:

1. Convert every character into lower-case,
2. Mask non-alphanumeric characters by white-space; i.e., any character other than a-z, A-Z, or 0-9 should be replaced by a single-space character,
3. Consider a word at the end of a line and the word at the start of the next line as non-adjacent; i.e. we do not consider this a co-occurrence.

Your program will output the graph as an adjacency list in a file "output.txt." Your output file must follow the following format:

```
predecessor-word1 count-of-outgoing-edges
<successor-word1, weight>
<successor-word2, weight>
...
predecessor-word2 count-of-outgoing-edges
<successor-word1, weight>
<successor-word2, weight>
...
```

The actual order of your entries for predecessor words, and among their successor words doesn't matter. So long as the correct successor words are listed underneath their predecessor with the correct weight.

1.1 Example

Suppose we give your program the following (true) statement:

It is raining. And, it is not raining.
Therefore, George Washington's body is made of raining rakes.

Following the requests of the last section, we convert this to,

it is raining and it is not raining
therefore george washington s body is made of raining rakes

Then, the output file should be something like:

```
it 1
<is, 1>
is 3
<raining, 0.333>
<not, 0.333>
<made, 0.333>
raining 2
<and, 0.5>
<rakes, 0.5>
and 1
<it, 1>
not 1
<raining, 1>
```

```
therefore 1
<george, 1>
george 1
<washington, 1>
washington 1
<s, 1>
s 1
<body, 1>
body 1
<is, 1>
made 1
<of, 1>
of 1
<raining, 1>
```

2 Creating a Spark Environment

To help with your implementation, a Docker image has been created with Spark and other relevant dependencies. Here, we detail the steps for setting up the Docker image and running the code template in a Docker container.

Note: It is not compulsory to use the Docker image to work on the assignment. You are free to download and install Spark directly by following the instructions on the Apache Spark website

2.1 Installing Docker

The first step is to install Docker and get it running on your system. Below are generic installation instructions for Docker. Check out the official Docker Website for detailed steps.

Linux Platforms For Linux platforms, install the Docker Engine by following the instruction for the appropriate OS.

Mac For a Mac, install Docker Desktop

Windows For a Windows system, install Docker Desktop.

2.2 Verifying the Docker Installation

To verify that the installation was successful, open a new terminal and run the following command

```
docker image ls
```

If Docker was installed successfully, you should see an empty table of images.

```
REPOSITORY  TAG       IMAGE ID  CREATED  SIZE
```

2.3 Pulling the Docker Image

Now you can try pulling the image. The Docker image you'll be using for this assignment is `utece360p/hw4`. This Docker image contains all the Spark and Java dependencies needed to complete this assignment.

To pull this image, run:

```
docker pull utece360p/hw4
```

Verify that the docker image was pulled successfully by running the following command

```
docker image ls
```

This time, you should see one image in the table of images.

```
REPOSITORY                TAG       IMAGE ID       CREATED       SIZE
utece360p/hw4   latest       ...           ...           ...
```

2.4 Starting a Docker Container

Navigate to the root directory of the template for assignment 4 in the git repository. Your current directory structure should look like this

```
|
 -- src/
```

Now you can try to spin up a container for this image using the below command after replacing the path to your local src directory. (For Windows systems, you may need to provide the absolute path to src inside quotation marks.)

```
docker run -p8080:8080 -p18080:18080 -it \
-v <path_to_local_src_directory>:/src \
utece360p/hw4
```

This command spins up a container running the specified Docker image. It mounts your local directory `./src` onto the container's file system. The `-p` options map ports from the docker container to the host machine. (You can map in more ports depending on your requirement.) The docker container is opened in an interactive mode. When you need to close the container, run the `exit` command.
On initialization, the container searches for the file `/src/start_spark.sh` in the mount point and executes it. This script essentially starts a Spark cluster- it deploys a Spark master node and 4 worker nodes. You can modify this script to try out different cluster configurations.
When the Spark nodes finish initializing, navigate to http://localhost:8080/. You should see the Spark Master UI with 4 worker nodes.

2.5 Submitting Tasks to the Container

Now that a Spark cluster is up and running in your docker container, you can try to submit tasks. Write up your solution in `WordGraph.java`.
When you're ready to test your code, navigate to the `/src` directory inside your container.

```
cd /src
```

The commands in `run_java.sh` can help you compile your java code into a jar and submit it to the Spark cluster.

```
./run_java.sh
```

Before beginning the assignment, you can test out the environment using one of Spark's example executables. The Spark example that calculates the value of Pi can be run using the following command

```
/opt/spark/bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://172.17.0.2:7077 \
--conf spark.eventLog.enabled=true \
/opt/spark/examples/jars/spark-examples_2.12-3.0.2.jar 10
```

2.6 Metrics Dashboard

The container also starts up a history dashboard with metrics of applications that have been successfully executed. Once you've run a Spark application you can check out the dashboard at http://localhost:18080/