Assignment 2: More MapReduce with Hadoop

Jean-Pierre Lozi

February 5, 2015

Provided files An archive that contains all files you will need for this assignment can be found at the following URL:

`http://sfu.ca/~jlozi/cmpt732/assignment2.tar.gz`

Download it and extract it (using "`tar -xvzf assignment2.tar.gz`", for instance).

1 Hadoop Streaming

We are now going to use the Hadoop Streaming API to write MapReduce programs in Python. If you are not familiar with Python, you can use another scripting language in this exercise (Ruby, PHP, etc.). Any language that can read from the standard input and write to the standard output works. The input file is written to the standard input of the Map function. The Mapper outputs key/value pairs as tab-delimited lines to the standard output. The Reducer reads tab-delimited key/value pairs from the standard input and also writes tab-delimited key/value pairs to the standard output. Here is an example of how *Word Count* would be written in Python:

File `wc_mapper.py`:

```
#!/usr/bin/env python

import sys

# We read from the standard input.
for line in sys.stdin:
    # Remove whitespaces at the beginning and the end of the line.
    line = line.strip()
    # Split the line into words.
    words = line.split()
    # Count!
    for word in words:
        # Write to stdout, using a tabulation between the key and the value.
        print '%s\t%s' % (word, 1)
```

File `wc_reducer.py`:

```
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# We read from the standard input.
for line in sys.stdin:
    # Remove whitespaces at the beginning and the end of the line.
    line = line.strip()

    # Split the line to extract the key and the value.
    word, count = line.split('\t', 1)

    # Since we're using the standard input, everything is text. We have to
    # convert the count variable to an integer (we ignore the value if it cannot
    # be parsed).
    try:
        count = int(count)
    except ValueError:
        continue

    # Hadoop sorts the output by key before it is parsed by the reducer.
    # Therefore, we know that we are counting the number of occurences of the
    # same word until the current word changes.
    if current_word == word:
        current_count += count
    else:
        # We are done for this word. Write the result to the standard output.
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# This is needed to output the last word.
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Question 1 You will find the `wc_mapper.py` and `wc_reducer.py` files in the archive provided with this assignment. Use them to count the words of the `gutenberg-100M.txt` text file. You will use the following

command:

```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
             -files wc_mapper.py,wc_reducer.py \
             -input gutenberg-100M.txt -output output \
             -mapper wc_mapper.py -reducer wc_reducer.py \
```

As a reminder, you can find the `gutenberg-100M.txt` file at the following location:

```
/cs/bigdata/datasets/gutenberg-100M.txt
```

The `-files` argument is needed to tell Hadoop to copy the Python files to remote nodes. Is the output the same as the one you obtained in the first assignment?

Question 2 Can we use a Combiner in this case in order to speed things up? How would you achieve this with a minimum amount of work?

Question 3 One of the advantages of the Hadoop Streaming interface is that we can test our scripts locally, in a single thread, without using Hadoop at all. All we need to do is to write the input file to the standard input of the mapper, to sort the result of the Mapper and to write the result to the standard input of the Reducer. How can we do that with a single command line in the console? What are the advantages and disadvantages of this approach.

Question 4 Modify `wc_mapper.py` so that it produces the same output as your *Word Count* program from the first assignment.

Question 5 We will now use the Hadoop Streaming API to work on the NCDC Weather data that we already used in the previous assignment. Like in the first assignment, we'll use the following file:

```
/cs/bigdata/datasets/ncdc-2013-sorted.csv
```

Using the Hadoop Streaming API, write a Mapper and a Reducer that will find the maximum of the minimum daily temperatures, and the minimum of the maximum daily temperatures, for each day. We want to use a single Reducer to solve the problem. You will disregard values below -79°C and above 59°C as being invalid. Unfortunately, this will not remove all invalid values, consequently, some of your results may look suspicious.

Question 6 The file `ghcnd-countries.txt`, provided with this assignment, associates country codes with countries:

```
AC,Antigua and Barbuda,
AE,United Arab Emirates,
AF,Afghanistan
AG,Algeria,
AJ,Azerbaijan,
...
```

Again, we will use the 2013 NCDC results. Remember, the file starts like this:

```
AE000041196,20130101,TMAX,250,,,S,
AG000060390,20130101,PRCP,0,,,S,
AG000060390,20130101,TMAX,171,,,S,
AG000060590,20130101,PRCP,0,,,S,
AG000060590,20130101,TMAX,170,,,S,
...
```

We want to join the two files, so that we know the number of lines of each type of record (`TMAX`, `TMIN`, `PRCP`...) we have for each country. The file `ghcnd-countries.txt` is incomplete, you will ignore the countries that are not found in that file (or create an `Unknown` category for them). The Mapper will take both files as its input (you can use `-input` twice), and output a list of values that include all relevant values for both files (some values will be undefined, depending on which file input comes from). The output will be designed so that lines that link country codes to country names will precede all NCDC data from that country after the Shuffle/Sort phase. The reduce will merge the data, relying on the ordering of its input.

Your output file should look like this:

```
United Arab Emirates    PRCP    157
United Arab Emirates    TMAX    296
United Arab Emirates    TMIN    226
Algeria PRCP    1433
Algeria TMAX    1190
Algeria TMIN    1126
...
```

2 MapReduce for Graphs

The objective of this exercise will be to use MapReduce to manipulate graphs. The dataset we will use is a graph of Wikipedia, in which nodes are pages, and their neighbors are the pages they link to. The dataset consists in two files:

```
/cs/bigdata/datasets/links-simple-sorted.txt
/cs/bigdata/datasets/titles-sorted.txt
```

The first file (`links-simple-sorted.txt`) contains the graph per se. Here are the first lines of the file:

```
1: 1664968
2: 3 747213 1664968 1691047 4095634 5535664
3: 9 77935 79583 84707 564578 594898 681805 681886 835470 880698 1109091 1125108 1279972
1463445 1497566 1783284 1997564 2006526 2070954 2250217 2268713 2276203 2374802 2571397
2640902 2647217 2732378 2821237 3088028 3092827 3211549 3283735 3491412 3492254 3498305
3505664 3547201 3603437 3617913 3793767 3907547 4021634 4025897 4086017 4183126 4184025
4189168 4192731 4395141 4899940 4987592 4999120 5017477 5149173 5149311 5158741 5223097
5302153 5474252 5535280
4: 145
...
```

From the first line, we can deduce that page #1 contains a single link to page #1,664,968. Page #2 contains links to pages #3, #747,213, #1,664,968, #1,691,047, #4,095,634, #5,535,664, and so on. The second file (`titles-sorted.txt`) contains the list of titles of all Wikipedia pages, with the title of page #*n* being located at the n^{th} line of the file.

2.1 Finding the shortest path

You will write a Hadoop program that finds the shortest path between two pages. It will take the identifiers of two pages as command-line arguments, and return the shortest path that goes the source page to the target page. For instance, going from page #2,111,690 ("Hadoop") to page #4,371,964 ("SFU_Vancouver") should give the following result:

```
"Hadoop" (2111690) -> "2008" (88822) -> "Canadian_federal_election,_2008" (899021) ->
"British_Columbia" (792600) -> "Simon_Fraser_University" (4605555) ->
"SFU_Vancouver" (4371964)
```

The algorithm will run a series of MapReduce jobs over the graph: the output of each job will be fed as the input of the next one. You will have to make sure to set an upper bound on the number of iterations (<10, for instance) so that in case there is a bug in your code, you don't end up with an infinite loop wasting the resources of the cluster.

The general idea of the algorithm is the following: the source page will initially have a distance of 0 while all other pages will have a distance of ∞. The first MapReduce iteration will make it so that all of the neighbors of the source page have a distance of 1. The second iteration will make it so that the neighbors' neighbors will have a distance of 2 unless they have already been reached with a distance of 1. At each iteration, the algorithm memorizes the path that was used to reach all nodes. When the targed node is reached, the algorithm stops and returns the shortest path that was used to reach it. This algorithm is illustrated in Figure 1.

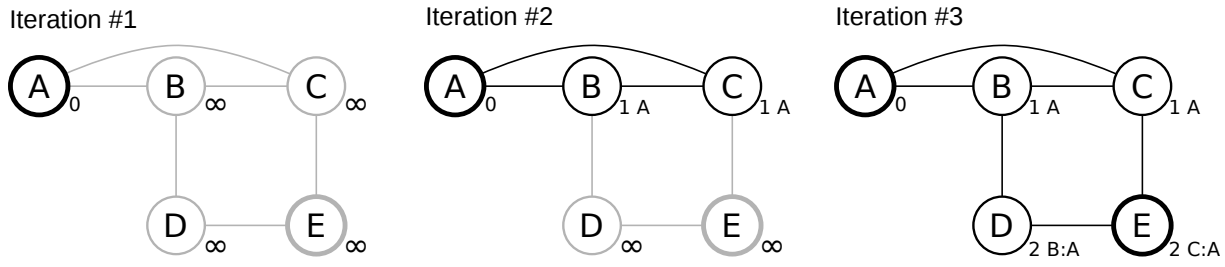

Figure 1: Algorithm for finding the shortest path (between A and E). Each iteration is MapReduce phase.

The Mapper receives the identifier of a page as an argument, and information about the page as its value, namely, the shortest distance, the corresponding partial path, and the list of its neighbors' identifiers. The pseudo-code for the Mapper is shown in Algorithm 1.

The Reducer first outputs the key and the value it receives so that the list of neighbors is not lost. If the node has been reached already, it updates the distance and the path of all of its neighbors, and outputs the corresponding key/value pairs.

The reduce() function is shown in Algorithm 2. The Reducer looks at all neighbors of a page that have been output by the Mapper, finds the one with the minimum distance, and outputs that neighbor with the path used to reach it.

function $map(page\_id,\ page = \langle distance,\ path,\ neighbor\_ids \rangle)$:
    output($page\_id,\ page$);
    if $distance \neq \infty$ **then**
        for $neighbor\_id \in neighbor\_ids$ **do**
            output($neighbor\_id,\ \langle distance + 1,\ page\_id\ .\ ":"\ .\ path \rangle$);
        end
    end

Algorithm 1: map() function

function $reduce(page\_id,\ [\_page, neighbor_1, neighbor_2, ...])$
    $distance \leftarrow \infty$;
    $path \leftarrow null$;
    for $x \in [\_page, neighbor_1, neighbor_2, ...]$ **do**
        if x is a node **then**
            $page \leftarrow x$;
        if $x.distance < distance$ **then**
            $distance \leftarrow x.distance$;
            $path \leftarrow x.path$;
    end
    $page.distance \leftarrow distance$;
    $page.path \leftarrow path$;
    output($page\_id, page$);

Algorithm 2: reduce() function

Here are a few tips to help you write the program:

- Start by creating a class that will represent a page, with its distance, partial path, and list of neighbors. Write a function that converts a text representation of the page into an instance of the class.
- Your program will first convert the input file into a temporary file that contains the text representation of each node that you defined. In this temporary input file, all nodes will have a distance of `Integer.MAX_VALUE` (∞), except the source node, whose distance will be 0. You can use the `KeyValueTextInputFormat` class to make it so that the Mapper will directly get the right key/value pairs from that input file.
- You will run a MapReduce job for each iteration, making sure that the output path of iteration n is used as the input path of iteration $n+1$.
- You can use a `Counter` object to communicate between the Reducer and the function that runs MapReduce jobs: when the Reducer detects that the shortest path has been found, it can write the shortest path and the corresponding distance to a `Counter`.
- The pseudo-code shown in Algorithm 1 and 2 only aims to explain the general idea of the algorithm. You may face problems that require workarounds when you implement these functions, it is normal if they are longer than the pseudo-code. For instance, the pseudo-code considers that all pages have an entry in the input file, whereas not all pages have an entry in `links-simple-sorted.txt`: pages that don't contain any links are only listed as neighbors.

- Finally, you will have to write a function that returns the title of a page given its identifier in order to output the shortest path between the source and the target page.

To debug your code, don't use the Wikipedia dataset at first: create an input file that contains a small graph with a few nodes, using the same format as `links-simple-sorted.txt`. When everything works, run your code on the Wikipedia dataset.

2.2 PageRank

You will now calculate PageRank over the Wikipedia graph. PageRank is the algorithm used by Google to rank its search results. The main idea of PageRank is that it gives a higher score to pages that have more inbound links. However, it penalizes inbound links that come from pages that have a high number of outbound links. Here is the formula PageRank uses for a node n:

$$PageRank(n) = \frac{1-d}{|graph|} + d \sum_{x \in inbound\_links(n)} \frac{PageRank(x)}{x.outbound\_links}$$

In this formula, d is the *damping factor*, for which we will use the value 0.85.

Each vertex in the graph starts with a seed value, which is 1 divided by the number of nodes in the graph. At each iteration, each node propagates its value to all pages it links to, using the PageRank formula. Normally, the iterative process goes on until convergence is achieved, but in order to simplify things and to avoid wasting the cluster's resources, in this exercise, we will only use five iterations.

The Mapper receives the identifier of a page as an argument, and information about the page as its value, as was the case in the shortest path algorithm. The pseudo-code for the Mapper is shown in Algorithm 3.

function $map(page\_id,\ page = \langle page\_rank,\ neighbor\_ids \rangle)$:
    output($page\_id,\ page$);
    $out\_page\_rank \leftarrow \frac{page.page\_rank}{|page.neighbor\_ids|}$;
    for $neighbor\_id \in neighbor\_ids$ **do**
        output($neighbor\_id,\ \langle out\_page\_rank \rangle$);
    end

Algorithm 3: map() function

The Reducer goes through adds up the values received from all neighbors, and uses the PageRank formula to update the PageRank of the current page. The pseudo-code of the reduce() function is shown in Algorithm 4. Here are a few tips to help you write the program:

- Like for the shortest path, you will need a class that will represent a page. What information will it contain?
- You will need to find out the number of nodes in the graph, i.e., the number of pages. To this end, you can just count the number of lines in the `titles-sorted.txt` file.
- Like for the shortest path, your program will convert the input file into a temporary file that contains the text representation of each node. This time, you will add the default value for the PageRank for each node ($1/|graph|$) to each entry.

function *reduce*($page\_id$, $[page, page\_rank_1, page\_rank_2, ...]$)
    $sum \leftarrow 0$;
    $page \leftarrow null$;
    for $x \in [page, page\_rank_1, page\_rank_2, ...]$ **do**
        if x is a node **then**
            $page \leftarrow x$;
        else
            $sum \leftarrow sum + x$;
        end
    end
    $page.page\_rank \leftarrow \frac{1-d}{|graph|} + d \cdot sum$;
    output($page\_id, page$);

Algorithm 4: reduce() function

- You will run a MapReduce job for each iteration, making sure that the output path of iteration n is used as the input path of iteration $n+1$.
- As was the case for Algorithms 1 and 2, the pseudo-code shown in Algorithms 3 and 4 only aims to explain the general idea of the PageRank algorithm. You may face problems that require workarounds when you implement these functions, it is normal if they are longer than the pseudo-code. For instance, the pseudo-code considers that all pages have an entry in the input file, whereas not all pages have an entry in `links-simple-sorted.txt`: pages that don't contain any links are only listed as neighbors.

Again, to debug your code, don't use the Wikipedia dataset at first: create an input file that contains a small graph with a few nodes, using the same format as `links-simple-sorted.txt`. When everything works, run your code on the Wikipedia dataset. What are the 10 pages with the highest PageRank? The five pages with the highest page ranks should be "United States", "2007", "2008", "Geographic coordinate system", and "United Kingdom".