

## The MapReduce Paradigm and MRJob

Due Thursday, November 2, 11:59 pm

### Learning Objective

From our class discussions you are familiar with the power of the MapReduce paradigm. It started off as part of the secret sauce that differentiated Google from other search engines of that time. In this assignment you will express the solution to a three problems using the MapReduce paradigm and the MRJob library. As we discussed in class, a MapReduce programmer needs to write a:

1. Mapper, which will transform a (key, value) pair into a different (key, value) pair and a
2. Reducer, which will collect all values emitted by a mapper with the *same key* and transform the incoming (key, [values]) to yet another (key, value) pair.

As we've discussed in class, the MapReduce infrastructure will align the emitted keys of a mapper with a reducer.

The beauty of this paradigm is that

- a. Individual map / reduce tasks are often quite simple
- b. We can create a flow of map/reduce tasks to express a more involved computation
- c. In addition to collecting values with the same keys (to be sent to a reducer), the infrastructure takes care of many details (e.g., sending tasks to various nodes, dealing with node failure etc.).

*This is an individual assignment. You are welcome to discuss high level conceptual ideas but the final code needs to be yours. Submissions will be checked for authenticity.*

### Assignment

The assignment is in similar vein to the exercises we did in class. You will want to start with the template code we've used in class.

#### 1. Solving Jumble

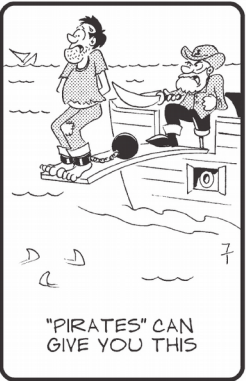
**JUMBLE**  
Unscramble these four Jumbles,  
one letter to each square,  
to form four ordinary words.

©2008 Tribune Media Services, Inc.  
All Rights Reserved.

www.jumble.com

Now arrange the circled letters  
to form the surprise answer, as  
suggested by the above cartoon.

THAT SCRAMBLED WORD GAME  
by Mike Argirion and Jeff Knurek



An example of a jumble word puzzle is given below. Given jumbled words such as VELGA, PLUIT, SICCUR, IMPAGE your program, `jumble.py` will print a list of 'unjumbled' words. To generate the anagrams you will use the official scrabble dictionary of words "sowpods"<sup>1</sup>. If interested check the Wikipedia entry for more details on sowpods. Be sure to delete the first two lines of the file you download so that only words are in the list. You will execute you program as below:

```
% python jumble.py jumble.txt sowpods.txt -q
```

`jumble.txt` scrambled words will be given one per line and tagged with a question mark, e.g.,  
`velga ?`

<sup>1</sup> <https://www.wordgamedictionary.com/sowpods/download/sowpods.txt>

```
pluit ?
```

A sample `jumble.txt` has been provide with several jumbled words. You output should give the solution (the unscrambled word) in some manner. For example, my output is along the lines of:

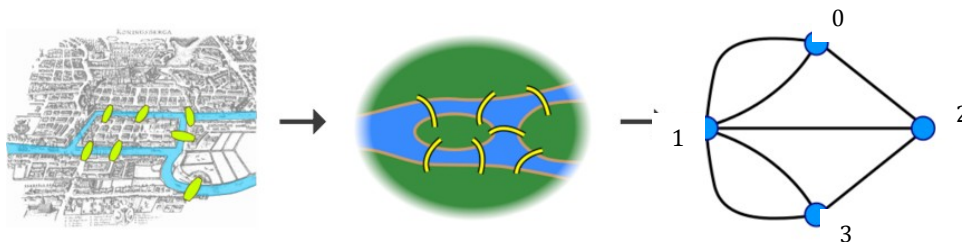
```
2      ["?velga", "gavel"]
3      ["?mursee", "emures", "resume"]
```

I'm using question marks in the output too (why?). The numbers at the beginning of each line are keys representing the length of the values emitted by the reducer (1 more than the number of anagrams). They don't play a role in the solution. You are welcome to use alternate approaches as long as the unscrambled words are clear.

Hint: Note that words that are anagrams of each other all have the same alphabetized letter ordering. For example, `spot`, `pots`, `tops` are anagrams and they all have the same letter ordering of `opst`. So in your `jumble.py` program you will write a mapper that yields the alphabetized version of a word and the original word. A reducer will then receive all the words that are anagrams of each other with the alphabetized ordering as the key. Note that the in the file `jumble.txt` we have marked each jumbled word with a '?' to differentiate it from words coming from the word list (`sowpods.txt`).

## 2. Eulerian Paths

One of the early, and still well known problem in graph theory, is determining an *Euler path*. The initial incarnation of this problem was studied by the renown mathematician Leonard Euler with the famous *Seven Bridges of Königsberg* problem. Following is a diagram from the Wikipedia article on this problem<sup>2</sup>. On the left you have a map of the seven bridges in the city of Königsberg. In the middle an abstract sketch and at the right a graph. The people of Königsberg pondered the question of whether they could walk across all seven bridges *without crossing a bridge twice*.



Euler proved that for a graph to have an Eulerian path all vertices of the graph need to have an even number of edges incident on it (i.e., the degree of all vertices is even). Write a MRJob program, which when given a graph will output just `True` or `False` based on whether it has an Eulerian path or not. Note that you will need two reducers for your solution. Take a look at the documentation for `mrjob`<sup>3</sup> to determine how to use two reducers (using `MRStep`). I've provided sample output of my program in the assignment folder. As I've shown, do produce a trace execution of your program so that it prints the degree of each vertex.

Create a graph called `konigsberg.txt` using the vertex numbering given above and run your program on it. My output is also included in the assignment folder.

---

<sup>2</sup> [https://en.wikipedia.org/wiki/Seven\\_Bridges\\_of\\_K%C3%B6nigsberg](https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg)

<sup>3</sup> <https://pythonhosted.org/mrjob/guides/writing-mrjobs.html#defining-steps>

### 3. Finding mutual friends

MapReduce is a useful paradigm for solving problems using graphs. Finding people who have mutual connections is a feature supported in social network systems. Suppose we have five people A, B, C, D, and E. The friends of each person are listed in the given ':' separated format (A is friends with B, C, and D; D is friends with A, B, C, and E etc.).

```
A : B C D
B : A C D E
C : A B D E
D : A B C E
E : B C D
```

Task: For a pair of friends, find their common friends.

The following guidelines may help:

- a. For each *pair of people*, list all their friends. The key will be the pair of friends in *sorted order*. For example line 5 of the input, E : B C D will result in the following key value pairs:

```
(B E) -> (B C D)
(C E) -> (B C D)
(D E) -> (B C D)
```

Note: Consider the first line. You interpret it as saying “somebody in the pair of people (B E) knows the people (B C D)”. You may be tempted to remove duplicate elements on the left and right. For example, in line 1 you may be tempted to remove the B on the right. It is easier to just keep it in.

- b. Common friends with the same key will be sent to the same reducer (by the ‘magic’ of the MapReduce framework). For example,

```
(A B) -> (C D)      # produced when processing line 1 of the input
(A B) -> (C D E)    # produced when processing line 2 of the input
```

will be sent to the same reducer and the input to the reducer will be  
key=(A B) values=((C D), (C D E))

- c. Now take the intersection of all the elements of values giving the common friends for (A B) i.e., (C D)  
Output for the given data set is below.

```
A B -> C D
A C -> B D
A D -> B C
B C -> A D E
B D -> A C E
B E -> C D
C D -> A B E
C E -> B D
D E -> B C
```

Output of the mapper:

```
['A', 'B'] -> ['B', 'C', 'D']
['A', 'C'] -> ['B', 'C', 'D']
['A', 'D'] -> ['B', 'C', 'D']
```

```
['A', 'B'] -> ['A', 'C', 'D', 'E']
['B', 'C'] -> ['A', 'C', 'D', 'E']
['B', 'D'] -> ['A', 'C', 'D', 'E']
['B', 'E'] -> ['A', 'C', 'D', 'E']
```

```
['A', 'C'] -> ['A', 'B', 'D', 'E']
['B', 'C'] -> ['A', 'B', 'D', 'E']
['C', 'D'] -> ['A', 'B', 'D', 'E']
['C', 'E'] -> ['A', 'B', 'D', 'E']
```

```
['A', 'D'] -> ['A', 'B', 'C', 'E']
['B', 'D'] -> ['A', 'B', 'C', 'E']
['C', 'D'] -> ['A', 'B', 'C', 'E']
```

`['D', 'E'] -> ['A', 'B', 'C', 'E']`

`['B', 'E'] -> ['B', 'C', 'D']`

`['C', 'E'] -> ['B', 'C', 'D']`

`['D', 'E'] -> ['B', 'C', 'D']`

Output of the mapper has been color coded to illustrate the transfer of data from the mappers to the reducer. For example the mapper output tagged in red (with the key `['A', 'B']`) goes to the same reducer. The reducer then takes the set intersection of `['B', 'C', 'D']` and `['A', 'C', 'D', 'E']` to produce the final answer of `['C', 'D']`

Output of the reducer (the final answer):

<code>["A", "B"]</code>	<code>["C", "D"]</code>
<code>["A", "C"]</code>	<code>["B", "D"]</code>
<code>["A", "D"]</code>	<code>["B", "C"]</code>
<code>["B", "C"]</code>	<code>["A", "D", "E"]</code>
<code>["B", "D"]</code>	<code>["A", "C", "E"]</code>
<code>["B", "E"]</code>	<code>["C", "D"]</code>
<code>["C", "D"]</code>	<code>["A", "B", "E"]</code>
<code>["C", "E"]</code>	<code>["B", "D"]</code>
<code>["D", "E"]</code>	<code>["B", "C"]</code>

Note: you do not need to produce trace output from the mapper. Just the final result from the reducer will suffice.

## What to Submit

A zipped file, e.g., a6-mrjob.zip, with three MRJob Python files: `jumble.py`, `euler_path.py`, `common_friends.py`, and one text file `konigsberg.txt`. Do not submit the graphs nor the `sowpods.txt` files.

Please test your submissions along the lines of

```
% python jumble.py jumble.txt sowpods.txt -q
% python euler_path.py g1.txt -q
  # similarly for g2.txt and konigsberg.txt (which you create)
% python common_friends.py friends.txt -q
```

## Grading Rubric

D	An attempt has been made but none of the parts of the assignment work correctly
C	One of three parts works but not the other two
B	Two parts of the assignment work but the third does not
A	All three parts work on given and additional examples.

+/- grades will be assigned for overall good development practices (well organized code; comments etc.)