CS205, Fall 2015
Computing Foundations for Computational Science
Ray Jones

HARVARD
School of Engineering
and Applied Sciences

Homework 1
Due October 4, 2015

This homework will be an exploration of the uses of Spark. In this assignment, you will run jobs locally (on your own machine or in the VirtualBox) as well as on Amazon AWS's Elastic MapReduce service.

Note: You should have signed up for AWS Educate as part of HW0, and received credit to run jobs on AWS. You should set up a billing alert (we suggest for $20) to make sure you don't accidentally use up your free credits without noticing.

Note: You should do problem 1 soon, to make sure that you are setup with Spark on Amazon AWS. Post on Piazza or in Slack if you encounter difficulties.

Get the HW1 skeleton files from GitHub **2**
Problem 1 - Set up Spark on AWS **3**
Problem 2 - Computing the Mandelbrot Set [10%] **4**
Implement the Mandelbrot computation 5
Change your implementation to balance work 5
Problem 3 - Anagram Solver [10%] **6**
Problem 4 - Graph processing in Spark [25%] **8**
Generating the Marvel Character Graph 8
Single-Source Breadth-First Search 8
Searching neighborhoods in the Marvel Universe 9
Relax diameter assumption, optimize for bandwidth 9
Problem 5 - Larger-scale Graph Processing with AWS & Spark[30%] **11**
Finding Directed Paths 11
Connected Components 11
Problem 6 - Markov Shakespeare[25%] **13**
Building the Model 13
Using the Model to Generate Text 13

Get the HW1 skeleton files from GitHub

To bring your repository up-to-date with the CS205 homework repository, we will add it as a remote, then use it as the basis for your work on HW1.

First, switch to your local repository directory on your machine (or VirtualBox). Most likely, you can do this with this command:

```
cd ~/cs205-homework
```

Then, add the CS205 class HW repository as a remote with the name "upstream":

```
git remote add upstream https://github.com/harvard-cs205/cs205-homework.git
```

Fetch the changes we've made to that repository:

```
git fetch upstream
```

Check out a HW1 branch based on the upstream/master branch:

```
git checkout --no-track -b HW1 upstream/master
```

If you have uncommitted changes, you may have to deal with them. If you just want to throw them away, you can run "`git checkout -- .`", though you may just want to commit them all in a wrapup commit.

Make sure you are on the HW1 branch:

```
git branch
```

(You should see a list of branches with an asterisk next to `HW1`.)

Then, begin hacking, and `git add ...files...` and `git commit -m ‘‘...message...’’` as you go along.

When you are ready to submit your work, push it to GitHub into your remote repository (you can do this with partial results, if you want a remote backup of your work):

```
git push origin HW1
```

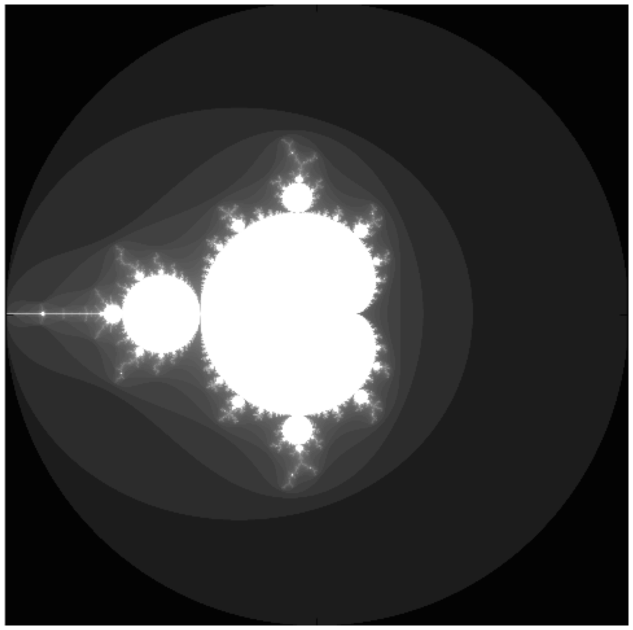
And then follow the instructions from Homework 0 to open a pull request from your HW1 branch to the CS205 homework repository.

Problem 1 - Set up Spark on AWS

As a large internet-based retailer, Amazon requires a powerful computing infrastructure to support its day to day operations. While others in a variety of fields could also benefit from having access to such computing resources, the costs associated with installing and maintaining a similar infrastructure often makes this infeasible, especially when the cluster would only be used sporadically. Amazon is targeting this market with their Elastic Compute Cloud (EC2) product, which we will use in this course.

Amazon AWS has recently made available Spark on their Elastic MapReduce framework (EMR is their Hadoop infrastructure, so does more than just MapReduce). We have created instructions for getting Spark running on an EMR cluster. Please use them to set this up on your VM (or local host, if you prefer), and make sure you can start a cluster, access it with ssh, and run pyspark. Run through the examples on Then shut your cluster down (for now).

Problem 2 - Computing the Mandelbrot Set [10%]



We will use the computation of the Madelbrot set (above) to explore how partitioning affects load balancing in Spark. The code for computing the Mandelbrot at a position (x, y) is:

```python
def mandelbrot(x, y):
    z = c = complex(x, y)
    iteration = 0
    max_iteration = 511  # arbitrary cutoff
    while abs(z) < 2 and iteration < max_iteration:
        z = z * z + c
        iteration += 1
    return iteration
```

In the image above, brighter pixels correspond to larger iteration counts.

An issue with parallelizing this computation is that the mandelbrot function can require anywhere from 0 to 511 iterations to return. In addition, as we can see in the image and will encounter below, locations close to one another tend to have similar final iteration counts.

Your task in the problem is to think about how you might partition (i.e., divide up) this task in Spark, predict the behavior of tasks (each corresponding to a partition) under such a partitioning strategy, and explore this behavior with the default partitioning and one of your own devising.

Load Balancing: In parallel computation with multiple subtasks, it is usually best for each task to be approximately equal in execution time. Slow tasks are known as "stragglers", and can cause the computation to have poor parallelism (particularly if badly scheduled). When you try to come up with an improved partitioning strategy, you should aim to have each partition/task take roughly the same amount of time.

Implement the Mandelbrot computation

Implement the computation of the Mandelbrot set, using the function above (provided in the `P2.py` file in the class HW repository). We've provided some helper functions, one for plotting the result (`draw_image`), and another for exploring how much computation was done in each partition (`sum_values_for_partitions`).

You should use the following constraints:

- Compute an image of 2000×2000 pixels.
- For the pixel at (i, j), use $x = (j/500.0) - 2$ and $y = (i/500.0) - 2$.
- Divide the computation where you call `mandelbrot` into 100 partitions.
- Use the default partitioning strategy (i.e., do not use `partitionBy()`.

You may find the `rdd1.cartesian(rdd2)` function useful, though be aware that the resulting RDD is partitioned into $N \times M$ partitions if passed RDDs with N and M partitions, respectively.

Use `sum_values_for_partitions` to produce a histogram of compute "effort" on each partition. (`plt.hist() and plt.savefig()` will be useful for this part of the problem.)

Change your implementation to balance work

Modify your code to more equally balance the work across partitions/tasks. Make sure you still use 100 tasks for the Mandelbrot computation.

There are several possible partitioning strategies to achieve this goal, with different tradeoffs. Produce a histogram of compute effort for each partition for your partitioning choice. Discuss your choice in terms of (a) how well it balances work, and (b) its costs, particularly shuffle costs.

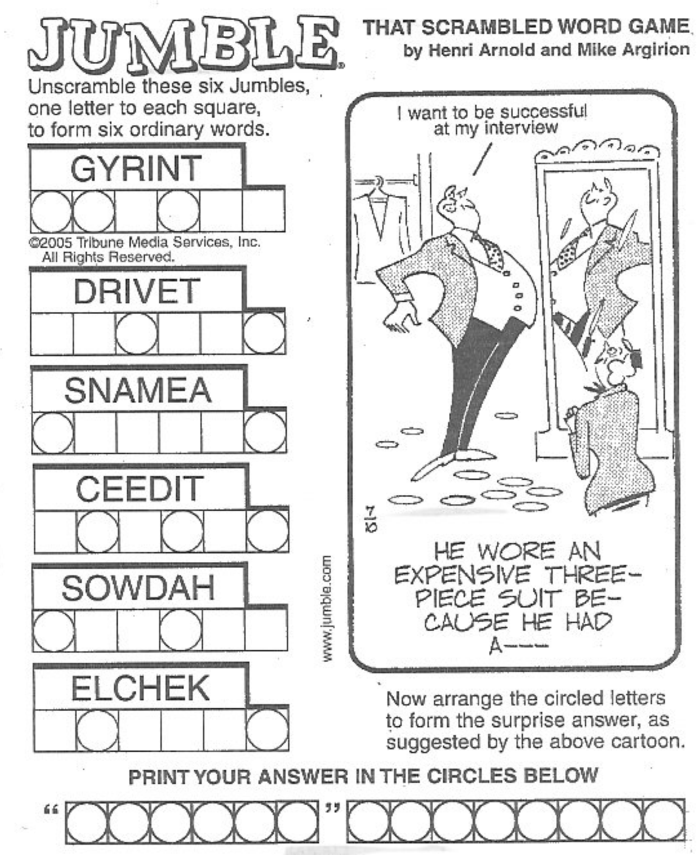
You may find it useful to explore your empirical results in the Spark UI, which is usually available at `http://localhost:4040` when running locally.

Submission Requirements

- `P2a.py`: Completed pyspark script for plotting the Mandelbrot function with default partitioning.
- `P2a_hist.png`: Your histogram of per-partition work for P2a.py.
- `P2b.py`: Completed pyspark script for plotting the Mandelbrot function with load-balanced partitioning.
- `P2b_hist.png`: Your histogram of per-partition work for P2b.py.
- `P2.txt`: Your discussion of the results and tradeoffs.

Problem 3 - Anagram Solver [10%]

Jumble is a common newspaper puzzle in the United States. As seen below, it consists of a series of anagrams that must be solved. Circled letters are then used to create an additional anagram, whose solution answers a question posed by a small cartoon. In especially difficult versions, some of the anagrams in the first set can possess multiple solutions. The correct solution can only be determined by trying to solve the subsequent cartoon. Therefore, when solving this puzzle, it can be quite important to know all possible anagrams of a given series of letters. In this problem, we're going to compute all possible anagrams for all possible words.


JUMBLE
THAT SCRAMBLED WORD GAME
by Henri Arnold and Mike Argirion
Unscramble these six Jumbles, one letter to each square, to form six ordinary words.
GYRINT
©2005 Tribune Media Services, Inc. All Rights Reserved.
DRIVET
SNAMEA
CEEDIT
SOWDAH
ELCHEK
www.jumble.com
I want to be successful at my interview
HE WORE AN EXPENSIVE THREE-PIECE SUIT BE-CAUSE HE HAD A---
Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.
PRINT YOUR ANSWER IN THE CIRCLES BELOW

You can download a word list (taken from The English Open Word List from S3 on AWS. If you are running Spark on AWS, you can access it directly using

```python
wlist = sc.textFile('s3://Harvard-CS205/wordlist/EOWL_words.txt')
```

Using pySpark and the provided word list write a Python script that generates sets of valid anagrams for all words in the word list. Generate an RDD of the form:

```
(SortedLetterSequence1, NumberOfValidAnagrams1, [Word1a, Word2a, ...])
(SortedLetterSequence2, NumberOfValidAnagrams2, [Word2a, Word2b, ...])
...
```

where `SortedLetterSequence` is a sequence of letters in alphabetical order (and is distinct from any other `SortedLetterSequence`), `NumberOfValidAnagrams` is the number of words that can be constructed from the letters in `SortedLetterSequence`, and the list at the end is the list of words that can be constructed using all of the letters in `SortedLetterSequence`.

You should not need to tune Spark (i.e., change partitioning count or strategy) for this task, or use AWS (though you may), as the data file is not that large, and the computation should take only a short time to complete.

Your script should extract and print the line from the RDD above with the largest number of valid anagrams.

Submission Requirements

- `P3.py`: Complete script solving the anagram exercise
- `P3.txt`: A single entry from the RDD, namely the entry with the most anagrams. **HINT:** For our dictionary, the entry with the most anagrams should have 11 anagrams.

Problem 4 - Graph processing in Spark [25%]

Graphs are ubiquitous in modern society. Examples encountered by almost everyone on a daily basis include the hyperlink structure of the web, social networks (manifest in the flow of email, phone call patterns, connections on social networking sites, etc.), and transportation networks (roads, bus routes, etc.). Search algorithms on graphs are invoked millions of times a day, e.g., whenever anyone searches for directions on the web.

For this problem, we will utilize the characters found in the Marvel Comic universe and a graph formed from those characters who appear in the same comic issues.

Generating the Marvel Character Graph

Alberich, Miro-Julia, & Rossello (2002) examined the characteristics of the graph formed by the characters and comics in the Marvel Comic universe, and for this problem we will be using the data that inspired their work. Begin by downloading their data file to your local system (or VirtualBox). Each line in this file contains a "CHARACTER NAME", "COMIC ISSUE" pair (with quotes), where the existence of a pair implies that that character appeared in the associated comic. We can conceptualize these relationships as a graph, where each character is represented by a node and an edge exists between any two nodes if the corresponding characters appear in the same comic issue.

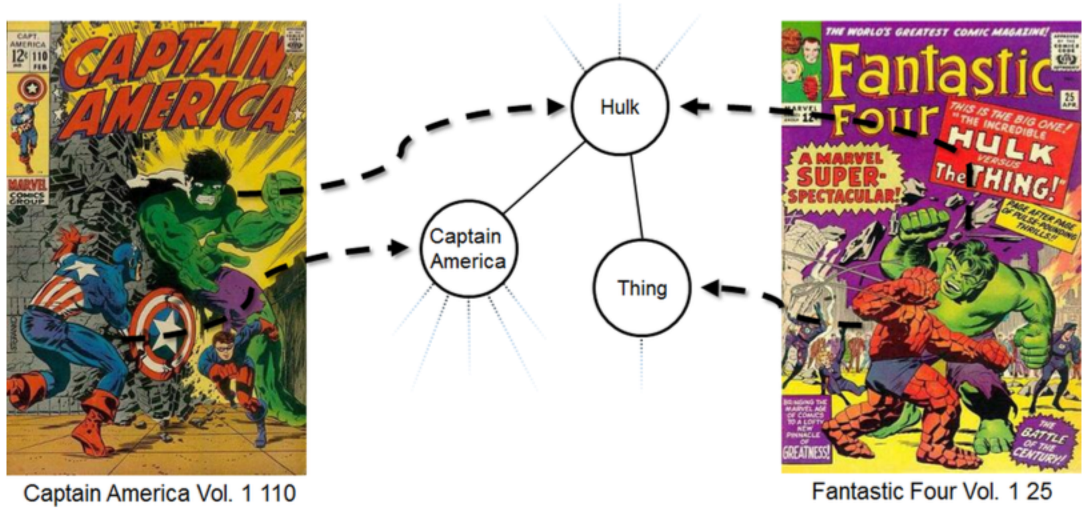


Figure 1: Interpreting character associations in a comic corpus as a graph

Your first task will be to convert this source data into a graph representation of the links between characters (or nodes). You can freely choose your representation in Spark, but be sure that whatever your choice, links between characters are symmetric.

Single-Source Breadth-First Search

One of the most common and well-studied problems in graph theory is the single-source breadth-first search (SS-BFS) problem, where the task is to find the shortest paths from a source node to all other nodes in a graph. Given the graph of characters you created, you will now execute a single-source shortest-path algorithm starting from a given root node to determine the shortest distance to all other (connected) nodes in the graph.

To perform this search, you will use an iterative approach, where each iteration expands the search frontier by one hop. To start, the root node is set to distance 0. In the first iteration, neighbors of the root node will be set to distance 1. In the second iteration, neighbors of the distance 1 nodes will be set to distance 2 (except nodes that already have a distance < 2). In the third, neighbors of nodes at distance 2 are assigned distance 3 (except those already < 3), and so on.

For the purposes of this problem, you may assume that the diameter of this graph is ten (10). What does this diameter imply about the maximum number of steps in which your search must be executed? In practice, determining a graph diameter is difficult, both from a theoretical and technical (q.v. HADI diameter estimation) perspective.

Searching neighborhoods in the Marvel Universe

Execute your SS-BFS job three separate times using your prepared Marvel graph for the source vertices listed below. Note that between iterations, you should make sure to use new distances.

- ”CAPTAIN AMERICA”
- ”MISS THING/MARY”
- ”ORWELL”

Count the number of nodes touched in each of the above searches and place these counts in your writeup. What does it mean if a character does not have a defined distance during the search?

Relax diameter assumption, optimize for bandwidth

Using Spark Accumulators, remove the assumptions on the diameter of the graph.

In addition, use `rdd.filter()` to optimize the amount of data transferred during each update, noting that only some nodes contribute to the update. Be aware of shuffles required by your code, and ensure that any that are required, are as small as possible.

Submission Requirements

- `P4_bfs.py`: Spark code implementing breadth first search, given a graph RDD and a character name, using the optimized version from the last part.

- `P4.py`: Wrapper code that loads the data, creates the graph RDD, and uses the functions you write in P4_bfs.py to search for the given nodes above.
- `P4.txt`: A description of your graph representation, and the number of touched nodes in each of the searches above. Also, discuss the two questions above (re. graph diameter, and untouched nodes).

Problem 5 - Larger-scale Graph Processing with AWS & Spark[30%]

We have copied a dataset of page names and links extracted from Wikipedia in 2010 to S3. The format of these files is documented by the author, Henry Haselgrove.

The two files can be accessed on AWS using

```
links = sc.textFile('s3://Harvard-CS205/wikipedia/links-simple-sorted.txt')
page_names = sc.textFile('s3://Harvard-CS205/wikipedia/titles-sorted.txt')
```

You may want to develop the code below using the Marvel graph data, above, and then copy and adjust it for AWS.

Note that the page names and links are 1-indexed!

Also note that the links dataset is about 10-times larger than the pages dataset; use this to choose a partitioning strategy that makes your code below more efficient.

Finding Directed Paths

Extend your work from the Marvel Graph BFS problem to find shortest paths in the Wikipedia dataset between two nodes, and report such paths.

Note that the Wikipedia graph is directed, i.e., a link from A to B does not imply a link from B to A.

Use your code to compute one of the shortest paths from "Kevin_Bacon" to "Harvard_University", and the reverse (Harvard to Mr. Bacon). Report these results in your writeup below[1].

Connected Components

The Connected Component problem is that of identifying sets of nodes in a graph that are linked via some path.

Implement an algorithm to find connected components within two graphs derived from the Wikipedia graph: the graph where each link is taken as symmetric, and the graph where two pages (i.e., nodes) are only considered linked if they link to each other.

Report the number of connected components in each of these derived graphs, and the number of nodes in the largest connected component (in your writeup).

Hint: this can be phrased as an iterative computation involving `min()`.

[1]Style points will be awarded for finding a more interesting pair to find paths to/from.

Extra Credit Implement and explain how you can compute the number of connected components in $O(\log D)$ iterations, where D is the diameter of the graph.

Submission Requirements If you include helper files/functions, (like `P4_bfs.py` above, please make sure they are included in the P5 subrirectory of your work.

- `P5_bfs.py`: Script for finding shortest paths in the wiki data (suitable for AWS).
- `P5_connected_components.py`: Script for finding connected components in the wiki data (suitable for AWS).
- `P5.txt`: Writeup for the questions above (shortest paths and number of connected components.

Problem 6 - Markov Shakespeare[25%]

In this problem we would like you to build a simple statistical language model for the writing style of Shakespeare. Your model should be a simple Markov Chain of order 2. You will use that model to generate novel sentences based on Shakespeare's original texts.

Building the Model

Begin by downloading the complete works of Shakespeare from here. It is also available from S3 as:

`s3://Harvard-CS205/Shakespeare/Shakespeare.txt`

In the first step, you should parse the text file into words. Also, filter out any words that:

- contain only numbers,
- contain only letters which are capitalized,
- or contain only letters which are capitalized and end with a period.

Do not worry about filtering out punctuation or other symbols, or putting words in lowercase.

Note that we treat the remaining text in the file as a single, continuous stream of text. **Note: Be sure your filtering does not change the order of the words!**

After splitting into words and filtering, you should build an RDD that stores the following data:

```
((Word1, Word2), [(Word3a, Count3a), (Word3b, Count3b), ...])
```

Where `Count3a` is the number of times the phrase `''Word1 Word2 Word3''` appears **in order** in the original data.

For example, for the phrase "Now is ...", where `Word1 == ''Now''` and `Word2 == ''is''`, you should get the following values:

```
[('a', 1),
 ('be', 1),
 ('his', 1),
 ('that', 1),
 ('this', 1),
 ('it', 3),
 ('Mortimer', 1),
 ('the', 9),
 ('my', 2),
 ('your', 1),
 ('he', 1)]
```

Using the Model to Generate Text

To generate text from the model above, choose a random `(Word1, Word2)` from the RDD to start the phrase, and then choose `Word3` based on the counts in the model. **Your choice should be biased by the counts.** So, for example, when the model sees a phrase starting with "Now is" it should usually continue with "the", given the data above.

You may find the following functions useful:
`rdd.zipWithIndex()`, `rdd.lookup()`, `rdd.takeSample()`.

Note that there appears to be a bug in Spark 1.5.0 with `rdd.lookup()` on hash-partitioned RDDs, including those produced by `rdd.groupByKey()`. If you encounter this, you can fix it by inserting a no-effect map: `rdd.map(lambda x: x).lookup(...)`, but note that this breaks fast lookups. For this reason, do not worry about efficiency in this problem.

Write a script to generate 10 random phrases from the model, each with 20 words. Include these sentences in your writeup.

Submission Requirements

- `P6.py`: Completed python script to build the model and generate new phrases.
- `P6.txt`: Random phrases generated by your model.