CMS/CS/EE 144 **Guru: Yujie/Xander**

HW 4: PageRank & MapReduce

Assigned: 1/27/2016 Due: 2/5/2016 at 9:00am

We encourage you to discuss these problems with others, but you need to write up the actual solutions alone. At the top of your homework sheet, please list all the people with whom you discussed. Crediting help from other classmates will not take away any credit from you.

You may turn in a hard copy of your assignment or email your assignment to cms144.caltech@gmail.com. If you turn in a hard copy, ***remember to keep a copy of your homework to use in your self-evaluation****. If you turn in an electronic copy,* ***be sure to submit it as a single file****.*

Start early and come to office hours with your questions! We also encourage you to post your questions on Piazza, as well as answer the questions asked by others on Piazza.

0 Rankmaniac

Form a group of 2-4 and choose a team name. Have one of your group members send an email to the TAs at cms144.caltech@gmail.com by the end of next Wednesday (Feb. 3) with your group details. No group of 5 or more than 5 people is allowed.

1 Warmup with PageRank and stationary distributions [10 points]

In class we saw that PageRank can be viewed as calculating the stationary distribution of a transition matrix defined by a graph. In this problem we'll investigate the differences between various ways of calculating the stationary distribution. In particular, we have seen two ways of calculating the stationary distribution of a transition matrix P. One is the iterative method $\pi = \lim_{n\to\infty} \pi_0 P^n$ for some initial π_0, and the other one is to solve the equation $\pi = \pi P$.

For the following probability transition matrices, first use $\pi = \pi P$ to get the stationary distribution, and then show whether or not $\pi_0 P^n$ converges as $n \to \infty$. If it converges, show that $\pi = \lim_{n\to\infty} \pi_0 P^n$ does not depend on π_0 and interpret what this means about the stationary distribution. (Hint: you may want to diagonalize P to handle P^n easily.)

(a) $\begin{pmatrix} 0 & 3/8 & 5/8 \\ 2/3 & 1/4 & 1/12 \\ 4/9 & 0 & 5/9 \end{pmatrix}$

(b) $\begin{pmatrix} 0 & 1/4 & 0 & 3/4 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 3/4 & 0 & 1/4 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

2 Training to be a farmer [25 points]

A typical way to raise the PageRank of a page is to use "link farms", i.e., a collection of "fake" pages that point to yours in order to improve its PageRank. Our goal in this problem is to do a little analysis of the design of link farms, and how their structure affects the PageRank calculations.

Consider the web graph. It contains n pages, labeled 1 through n. Of course, n is very large. As mentioned in class, we use the notation $G = \alpha P + \frac{1-\alpha}{n}(\mathbf{1}_{n\times n})$ for the transition matrix. Let r_i denote the PageRank of page i, and $r = (r_1, r_2, \cdots, r_n)$ denote the vector of PageRanks of all pages.

Note: For a page that has k outgoing links, we put $1/k$ for the corresponding entries of P. However, when a webpage has no outgoing links, we add a 1 as the corresponding diagonal element of P for making its row-sum one. Note that this makes G a valid transition probability matrix.

(a) You now create a new web page X (thus adding a node to the web graph). X has neither in-links, nor out-links. Let $\tilde{r} = (\tilde{r}_1, \tilde{r}_2, \cdots, \tilde{r}_n)$ denote the vector of new PageRanks of of the n old web pages, and x denote the new PageRank of page X. In other words, $(\tilde{r}_1, \tilde{r}_2, \cdots, \tilde{r}_n, x)$ is the PageRank vector of the new web graph.

Write $\tilde{r}$ and x in terms of r. Comment on how the PageRanks of the older pages changed due to the addition of the new page (remember n is a very large number). Hint: Use the stationary equations to calculate PageRank, not the iterative approach.

(b) Unsatisfied with the PageRank of your page X, you create another page Y (with no in-links) that links to X. What are the PageRanks of all the $n + 2$ pages now? Does the PageRank of X improve?

(c) Still unsatisfied, you create a third page Z. How should you set up the links on your three pages so as to maximize the PageRank of X?

(d) You have one last idea, you add links from your page X to older, popular pages (e.g.: you add a list of "Useful links" on your page). Does this improve the PageRank of X? Does the answer change if you add links from Y or Z to older, popular pages?

(e) Describe what steps you might take to raise the PageRank of X further. You do not need to prove anything here, just summarize your thoughts based on the previous parts. For extra credit though, you can prove what the structure for a link farm with m nodes should be to optimize the PageRank of X.

3 Beyond PageRank [25 points]

PageRank is one example of a definition of importance, a.k.a., *centrality*, that is used to capture the relative importance of pages the web graph. Centrality is a much more general concept however, and is useful broadly in networks. For example, we may be interested in knowing how important a person is in a social network, or how important a road is in a city's transportation infrastructure.

As you might expect, there are a number of different ways of defining the centrality of a node in a network, and each might induce a different ranking of the nodes in terms of "importance". How appropriate any one definition is depends on the application scenario. In this exercise, we'll introduce you to the four most common measures of centrality.

Let's start with the definitions. Consider a **connected**, **undirected** graph G with n nodes, labeled $1, 2, \cdots, n$.

1. **Degree centrality** Let d_i denote the degree of node i. The degree centrality of node i is defined by

$$C_D(i) = \frac{d_i}{n-1}.$$

This definition of centrality assigns "importance" to a node proportional to its degree.

2. **Closeness centrality** Let $l(i,j)$ denote the distance (length of the shortest path) between nodes i and j. The closeness centrality of node i, denoted $C_C(i)$, is defined as the reciprocal of the average distance between i and all other nodes, i.e.,

$$C_C(i) = \frac{n-1}{\sum_{j \neq i} l(i,j)}.$$

According to this definition, a node that has, on average, shorter paths to other nodes is "more important".

3. **Betweenness centrality** Let $P(j,k)$ denote the total number of shortest paths between nodes j and k, and $P_i(j,k)$ denote the number of those shortest paths that pass through i. One can think of $\frac{P_i(j,k)}{P(j,k)}$ as a measure of the "importance" of i with respect to connecting j and k. Betweenness centrality attempts to capture how "important" a node is with respect to connecting other nodes in the graph. The betweenness centrality of node i is defined as

$$C_B(i) = \frac{\sum_{j,k:j \neq k, j,k \neq i} \frac{P_i(j,k)}{P(j,k)}}{\binom{n-1}{2}}.$$

4. **PageRank** Suppose you replace each undirected edge (i,j) in G by two directed edges $(i \to j)$ and $(j \to i)$. This gives us a directed, strongly connected graph. On this graph, we define the centrality of a node to be its PageRank, obtained by using the first cut PageRank algorithm you've seen in class (i.e., with $\alpha = 1$). Recall that the vector of PageRanks $r = (r_1, r_2, \cdots, r_n)$ obtained using this algorithm will satisfy

   (i) $r_i \geq 0$ for all i,

   (ii) $\sum_{i=1}^{n} r_i = 1$,

   (iii) $r_i = \sum_{j \in N(i)} \frac{r_j}{d_j}$ for all i.

   Here, $N(i)$ is the set of neighbors of node i and d_i is its degree.

Phew! With the definitions out of the way, let's get what you have to do.
Your task:

(a) Contrast the four definitions of centrality described above. Specifically, for each pair of definitions, either prove that for *any* connected, undirected graph G, the two definitions rank the nodes in the same order of importance, or give a counterexample that proves otherwise.

Hint: To get you started, we'll tell you that the PageRank r_i we've described turns out to be proportional to the degree centrality $C_D(i)$. Of course, you still have to prove this! (Don't be confused though, what we're claiming is that for an *undirected* graph, PageRank is equivalent to degree centrality. This is certainly not true for the web graph, which is *directed*.)

(b) For each of the centrality measures, describe a specific application setting where it best captures the notion of "importance" for the nodes.

4 MapReduce: Warming up for rankmaniac [40 points]

The idea:
You have a problem that involves a lot of computation on a large dataset. Now, imagine you own a datacenter. You have a large number of servers at your disposal. How would you go about leveraging your assets? Think of the simplest (possibly the dumbest) way to solve your problem. It's divide and conquer. Divide your computation into a large number of chunks. If the chunks are logically independent, i.e., they do not depend on each other to execute, then run these operations in parallel on different servers and aggregate the results to give a meaningful output. In 2004, Jeffrey Dean and Sanjay Ghemawat from Google proposed a framework called *MapReduce* to do exactly the same. This sounds simple but its implementation has found its way into a lot of real-world distributed computing scenarios. In fact, MapReduce was used by Google to recompute Google's index or PageRank for the web... and this is exactly what you'll be doing this year for Rankmaniac. Through this exercise, we'll get you ready for rankmaniac by exploring how to use MapReduce to solve a few problems.

Let's start with an example: *WordCounter*. Our goal is to count the number of occurrences of different words in a million files. First, we present the pseudocode and then discuss in detail how this actually works.

```
# Example: WordCounter

function main(){
        # Here we call the MapReduce in a
        # suitable way to solve our problem.

        # Define the input for MapReduce.
        listFilenames[] = list of Filenames.
        listKeyValues[] = list of pairs (filename, fileContents)
                where fileContents is a string containing the
                contents of the file with name 'filename'.

        # Run MapReduce on the list listKeyValues.
        listOfOutputs = MapReduce(listKeyValues)

        # Output result
        Print contents of listOfOutputs on console.
}

function MapReduce(listKeyValues){
        # This function defines the MapReduce framework.

        # Map Step:
        For each (key, value) pair in listKeyValues, run Map(key, value)
                # Each Map() operation 'emits' a new (key, value) pair
                # as output.

        # Collector Step:
        Collect all outputs from Map() in the form of (key, value)
                pairs in a new list called listOfMapOutputs.
                # This step runs in parallel with the Map
                # operations and catches all (key, value) output
                # pairs to create the list 'listOfMapOutputs'.
        Make a list of distinct keys in listOfMapOutputs.
        For each unique key, create a list of the values
                to create (key, listOfValues) pairs.
```

```
        # Reduce Step:
        For each such (key, listOfValues), run Reduce() on them.
                # Note that each distinct key goes to one Reduce()
                # call. This step also happens in parallel.

        # Return the results:
        Concatenate the results from all calls to Reduce() into a
                list and return it.
}
function Map(filename, fileContents){
        # For the input, key = filename, value = fileContents.

        words[] = Split fileContents by " ".
        for each word in words{
                EmitIntermediate(word, 1)
                # Note that Map() keeps on throwing out results.
                # Notice the difference from a 'return' statement.
        }
}
function Reduce(word, listValues){
        # For the input, key = word and value = a list of 1's.
        # As explained below, no. of ones for each word is the
        # no. of occurrences of the word in all files.
        # And hence...
        return (word, length(listValues))
        # Key = word (It has to be the same as the input key).
        # Value = No. of occurrences of word in all the files.
}
```

Let us see the way the framework operates. First, main() invokes the MapReduce on a list of (key, value) pairs. Internally it calls the Map function on each of these pairs. Different calls to Map() can potentially run on different machines in parallel and it keeps emitting (key, value) pairs as outputs by calling EmitIntermediate(). The Collector step runs in parallel and collects all such (key, value) pairs coming from the EmitIntermediate() operations. After all the Map operations are completed, these (key, value) pairs are grouped by distinct keys. For each unique key, we make a list of all values corresponding to that key to create a list of (key, listOfValues) pairs. Now run Reduce on each such (key, listOfValues) to compute a (key, value) pair as output. The specific computation depends on the application. Also, Reduce() operations can run in parallel, potentially on different machines. Finally a list is made out of the outputs of the Reduce operations and passed back to the main(), which then creates a meaningful output from it. Note that inputs and outputs of Map() and Reduce() are in the form of (key, value) pairs. The keys and values can be any data-structures (e.g. the "value" input to Reduce will be a list of values emitted).

The MapReduce function is already given in any implementation of this framework. It takes care of running Map, collect outputs and run Reduce. A programmer needs to write the main(), Map() and Reduce() steps for an application. And that's it! Let us see how WordCounter actually works.

- main() creates a list of (filenames, fileContents) pairs. Here fileContents is a string with the contents of the file.
- main() calls MapReduce and passes the above list to it.
- MapReduce takes each (filename, fileContents) pair from the list and runs Map() on it.
- The Collector step in MapReduce starts simultaneously that catches any output from the Map operations.

- Map() works on one file. It splits the text (in fileContents) into words. For each word, it emits the key/value pair (word, 1). If the word "amazing" occurs 3 times in the file `file23.txt`, the Map operation emits 3 pairs of ('amazing', 1) that are caught by the Collector. If `file24.txt` also has the word "amazing" in it, more such pairs are caught by the Collector. Eventually the number of such pairs caught by the collector will equal the number of occurrences of that word in all the files. Note that an alternative way of doing this would have been to actually count the number of occurrences of the word "amazing" in each file and return this in the value field, i.e., we could have chosen to output ('amazing', 3) from the Map operation on `file23.txt`. But that requires more computation than is necessary (the return result of Reduce() will have to be changed from length(listValues) to sum(listValues)).

- Collector collects all outputs and groups the results by unique keys. In WordCounter, each unique word corresponds to a unique key. Thus if the word "amazing" occurred only in files `file23.txt` and `file24.txt` as given before, there should have been four ('amazing', 1) pairs, which the collector concatenates into the pair ('amazing', $[1, 1, 1, 1]$). Now each such (key, list of values) pair is sent to a Reduce function.

- Reduce() gets one word and a list of ones. The number of ones in the list is exactly the number of occurrences of the word in all files. Thus we output (word, length of the list). For the word "amazing", we get ('amazing', 4).

- The results from all Reduce operations are merged into a single list that is sent back to the main().

- In our example, main() outputs the list of (word, # of occurrences) on the console.

In this assignment, we ask for the pseudocode to solve some problems with MapReduce. We found that writing pseudocode is sometimes trickier than coding in your favorite language. And job interviews often ask for pseudocode instead of the actual code. *This also serves as a "warmup" for Rankmaniac this year... where you'll be running MapReduce code on Amazon EC2.* So let's get started!

Note: Some MapReduce framework implementation may have different collector output (AKA Reduce() input). For example, Amazon Elastic MapReduce (EMR) feeds the Reduce() function with list of (key, value) pairs grouped by keys (each list will have the same key), instead of a single (key, listOfValues) tuple.

Problems to solve:

(a) ***Common friends [5 points]:*** One common processing request for social networking site like Facebook is the common friends feature. When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently so it'd be wasteful to recalculate it every time you visited the profile. Suppose Facebook have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. We're going to use MapReduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. As you are much more likely to visit your friend's page rather than someone you don't know, we are just going to preprocess common friends information among all pairs of friends.

Assume the friendship information is stored as a tuple (Person, [List of Friends]) for every person in the social network. You need to return a dictionary of common friends of the form ((A, B), [List of Common Friends of A and B]) for all pairs of (A, B) that are friends. The order of A and B within the tuple (A, B) should be the same as the lexicographical order of their names.

```
# Example: CommonFriends Template

function main(){

        listOfFriendship = list of (Person, [List of Friends])
        # Write your pseudocode here.
}

function Map(<key>, <value>){
        # Write your pseudocode here.
}

function Reduce(<key>, <listOfValues>){
        # Write your pseudocode here.
}
```

(b) ***High school days [10 points]:*** Suppose you have the test scores in Math from all high school students in California. These are stored in files where each file contains the test scores of one class. The scores are integers ranging from 0 to 100. Initially we have a list of filenames containing the test scores in unsorted order. Our job is to print all scores from all files in descending order. Implement this using MapReduce. Use the following template for the pseudocode and clearly state the keys and values for the inputs and outputs.

```
# Example: HighSchool Template
function main(){
        listOfFilenames = list of filenames containing
                test scores of all students.
        # Write your pseudocode here.
}

function Map(<key>, <value>){
        # Write your pseudocode here.
}

function Reduce(<key>, <listOfValues>){
        # Write your pseudocode here.
}
```

Hint: If you are thinking merge sort, think again!

(c) ***Good old π [10 points]:*** Suppose you have a random number generator that can generate uniformly from $[-1, +1]$. Call `rand()` to generate such a random number. Your goal is to approximately calculate the value of π using this. Describe in short a scheme how you would achieve this. Give the pseudocode for the Map and Reduce steps.
Hint: Befriend the circles and squares!
Note: Any probabilistic algorithm can be implemented using a similar approach.
Extra Credit: Consider the binary representation of π. Calculate the bits from position 1 million to position 5 million (to the right of the decimal point of π) using MapReduce. You do not need to write the pseudocode here. Just a few lines describing your technique will suffice.

(d) ***Gauge the distance [15 points]:*** You have an undirected weighted graph (all weights are positive) given by an adjacency list G, i.e., you are given a list where $G[i]$ is a list of (neighbor, distance) tuples of node i in the graph. G is read-only. Your goal is to find the shortest distances from node 1 to all other

nodes using MapReduce. Distance is the sum of the weights in the path. Use the following template and state your inputs and outputs clearly.

```
# Example: GaugeTheDistance Template

function main(){
        G[] = Adjacency list of the graph.
        # G[i] is a list of (neighbors, distance) tuple of node i.
        n = length(G)
        dist[] = list that will contain the distances
                from node 1 eventually. Initialize it arbitrarily.
        distUpdated[] = list that will contain distances from
                node 1 after each run of MapReduce in the
                following loop. Initialize it with (0,∞,∞,...,∞).

        while( NOT stoppingCriterion(dist, distUpdated)){
                dist = distUpdated
                # Clearly state the variables and fill in details.
                ...
                distUpdated = MapReduce(list of (<key>, <value>) pairs).
        }
        print the list dist[].
}
function stoppingCriterion(dist1, dist2){
        # Write your pseudocode here.
}

function Map(<key>, <value>){
        # Write your pseudocode here.
}

function Reduce(<key>, <listOfValues>){
        # Write your pseudocode here.
}
```

Extra Credit: Implement the stoppingCriterion using another MapReduce call.