1. **Introduction and Motivation**

Born from Google’s desire to create meaningful search results, PageRank is an important algorithm for determining the significance of nodes in a graph [13]. The general idea behind PageRank is simple, a node has a high PageRank if it has edges from other nodes with a high PageRank [13]. To apply this idea we can view the Internet as a directed graph with web pages for nodes and edges from web pages to the pages they link to. Then, a web page has a high PageRank if it is linked to by other pages with a high PageRank. Intuitively, this can be thought of as the fact that pages linked to by pages with high traffic will also have high traffic. By using this recursive idea, the PageRank algorithm can be used to assign some measure of importance to web pages. This algorithm is valuable to Google because they want to return the most relevant results for every search [13].
From a theoretical math standpoint, the PageRank problem involves some fairly straightforward linear algebra. We can calculate the PageRank vector using the power iteration method discussed later in the paper. However, the Internet is comprised of a huge number of webpages, so in practice a distributed algorithm is needed to compute PageRank in parallel and across data stored on many different machines. For this reason I focus on distributed algorithms for computing PageRank. Partitioning the work is more complicated, with many different approaches appearing in the literature. Many of these algorithms involve random factors, so analyzing these algorithms becomes a trickier game of proving convergence. Showing these algorithms are correct involves proving both that the algorithm approaches the correct answer, and that it does so in a feasible length of time.

Although the current research examines many different methods for calculating PageRank across a distributed system, there does not seem to be an empirical analysis of the performance of the algorithms compared against each other. Some papers compare their algorithm against the standard power iteration method, such as [1], but in the case of Avrachenkov et al. they do not seem to use strict speed as their metric of comparison. Instead they only look at the number of iterations required. An empirical analysis could tell us both the speed of each Algorithm and how the algorithms behave as the number of computers used scales up. This would help us decide which algorithm to choose when needing to rank the nodes of a huge graph stored on a cluster of computers.

Aside from ranking web pages, understanding PageRank gives us a way to extract meaningful data from any graph [20]. Graphs are a flexible and diverse data structure often used to represent networks of all types, so understanding algorithms for them is valuable. My goal is to understand these different algorithms, analyze what it takes to implement them in a distributed way, and compare the relative speeds both empirically and theoretically. This analysis can tell us which algorithms can be implemented in a scalable, distributed way, and which are actually worth implementing.

Algorithms were tested experimentally by running experiments on an Amazon Web Services cluster. AWS supplies clusters that can be run with distributed big data processing technologies such as Hadoop, MapReduce, and Spark on a platform known as Elastic MapReduce. Spark was the distributed computation platform used to test these algorithms. Experiments were run on this platform in order to understand the scalability and performance of these different algorithms.

2. Background

Before diving into PageRank, it is worth covering a little bit of background on linear algebra and Markov Chains. I use row vectors as opposed to the standard column vectors in order to stay consistent with the PageRank literature I encountered. This means that if \( v \) is a row vector of size \( n \), and \( M \) is an \( n \times n \) matrix, then \( vM \) is a vector of size \( n \), while \( Mv \) would be an \( n \times n \) matrix. This is the opposite of how column vectors work, but it should be clear from context.
Another useful tool is the eigenvector. An *eigenvector* of a matrix $M$ is a vector $\lambda$ such that $\lambda M = c\lambda$ for some real number $c$. This is useful because when $\lambda$ represents a probability distribution then if $\lambda$ is an eigenvector with eigenvalue $1$ it is essentially a probability distribution that remains unchanged by $M$, similar to an equilibrium.

A *Markov Chain* is a directed graph with weighted edges. The nodes represent different states of the Markov Chain, and the weights represent the transition probability of one state to another. That is, starting at state $i$ at time $t$, the edge $(i, j)$ has a weight corresponding to the probability of the Markov Chain being at state $j$ at time $t + 1$. This is a useful way to model the web because we can think of the web pages as states, links as edges, and the chance that a web surfer clicks on a link as the transition probability. A Markov Chain can be represented as a matrix that is identical to an adjacency matrix for a directed graph with weighted edges, we call this the transition matrix. This matrix will be important for calculating PageRank.

### 3. Outline of Prior Work

#### 3.1. Power Iteration

Given the importance of PageRank, and the success of Google, a wide array of research has already been completed. The PageRank algorithm can be seen as a subclass of the eigenvector centrality problem, which seeks to find the dominant eigenvector for the adjacency matrix of a graph [20]. In the case of PageRank, we can assume the matrix is row stochastic, that is all the values of the rows add up to 1, because the rows represent probabilities [20]. This assumption is key for many of the algorithms discussed in this paper. This includes the original paper by Brin et al, which describes how PageRank can be calculated using the power iteration method [13].

Their method relies on the fact that the probabilities of a web surfer ending up on any particular page can be represented as a vector with a number of entries equal to the number of pages. Since we can think of the hyperlink graph as a Markov Chain, we can think of the vector as the equilibrium reached by repeatedly applying the transition matrix of the chain, with a few alterations. In other words, the vector is an eigenvector of this transition matrix with eigenvalue $1$. Then, finding this eigenvector can be done using power iteration, which simply means multiplying an arbitrary vector by the transition matrix repeatedly. This is the same as taking a very high power of the matrix and multiplying it with a reasonable vector. Generally, the vector starts with equal probabilities for each page, so the value in each entry is $1$ divided by $n$, the number of nodes in the graph, *i.e.*, the number of web pages. Multiplying by the matrix can be seen as refining the vector until it is suitably close to the desired result. Therefore, our problem of ranking important nodes in a graph becomes a relatively straightforward matrix multiplication problem [13].

How do we know that this algorithm actually works? This algorithm can be proved correct by utilizing some helpful theorems from linear algebra and the theory of Markov Chains. These theorems tell us something about the transition matrix and how to figure out the equilibrium of a Markov Chain. I will cover a sketch of the proof based on the explanation found in [6] and the outline in [8].

Using a theorem from Norris, we know that a Markov Chain can reach equilibrium if it has an *invariant distribution*, is *irreducible*, and is *aperiodic* [12]. An *invariant distribution* is an
equivalent idea to an eigenvector with eigenvalue 1, a probability vector that does not change when multiplied by the transition matrix [12]. Then, we need to show that the transition matrix actually does have such an eigenvector. First, note that if a transition matrix for a Markov Chain had an eigenvector, it must have eigenvalue 1, since otherwise the values of the eigenvector would not add up to 1 after multiplying by the matrix. From a probability standpoint, this would mean that we could end up with a total probability different than 1, which does not make any sense. This fact can be confirmed by a quick check of the mechanics of matrix multiplication that takes advantage of the fact that our matrix is row stochastic. Next, we need to confirm that our matrix is guaranteed to have an eigenvector. To do so, we would normally have to find the characteristic polynomial of the matrix, but we can take a shortcut. Since the matrix has all positive entries, we can apply the Perron-Froebinius theorem to say that our matrix does indeed have an eigenvalue. The proof of this theorem is outside the scope of this paper, for a full proof see [11], but it guarantees that the matrix does have an invariant distribution.

Irreducible means that we can reach any node starting at any other node [12]. We know the matrix is irreducible from the fact that for any two pages, there is a chance that a web surfer will jump between them. This comes from our addition of equal probabilities for a web surfer to jump between random pages, which can be seen as the chance that a web surfer will get bored on a page, not click a link, and go to a different page. This value is most often referred to as $\alpha$. In the original papers by Google’s founders, they mention that they use $\alpha = 0.15$. They do not seem to provide any justification for this value, but they do note that more realistically each user would probably only have a few specific pages that they would randomly jump to instead of one value for all pages [3]. Langville notes that decreasing $\alpha$ provides a more accurate ranking, but can significantly increase the time taken to converge [6]. Thus, the $\alpha$ parameter is less about realistically portraying the actions of a web surfer, and more about finding a balance between capturing the structure of the web as accurately as possible and computing the PageRank vector in a reasonable amount of time. Also, this addition is essential for guaranteeing we can find the PageRank vector in the first place. With it, a web surfer can jump between any two pages, so our Markov Chain, and the corresponding transition matrix, is irreducible. [6].

Similarly, aperiodicity means that for any node and after any large length of time, there is a chance that our surfer will end up on that node [12]. This comes from the fact that our chance to jump between nodes includes a chance to refresh the current page, so any node is reachable after the first iteration [8]. Putting all of these concepts together, we are guaranteed that repeatedly multiplying by the transition matrix for our Markov Chain will indeed reach the invariant distribution.

To demonstrate the power iteration method, consider a basic web graph with 4 different pages and a link structure given by the following adjacency matrix.

\[
A = \begin{bmatrix}
0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0
\end{bmatrix}
\]

As a graph our matrix looks like this:
Now consider the matrix where each edge is weighted with the probability that a web surfer clicks on one link, assuming that for all links on a page a surfer gives equal consideration to all links. This gives us what Avrachenkov et al. call the hyperlink matrix $Q$.

$$
Q = \begin{bmatrix}
0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & \frac{1}{2} & 0
\end{bmatrix}
$$

However, we still need to add in our random chance for a surfer to jump to any page, $\alpha = 0.15$. Let $E$ be the matrix with all entries $1/4$, one over the number of nodes in our graph, then the PageRank matrix is given by:

$$M = \alpha E + (1 - \alpha)Q = \begin{bmatrix}
0.0375 & 0.3208 & 0.3208 & 0.3208 \\
0.0375 & 0.0375 & 0.8875 & 0.0375 \\
0.8875 & 0.0375 & 0.0375 & 0.0375 \\
0.4625 & 0.0375 & 0.4625 & 0.0375
\end{bmatrix}$$

Note that the rows of $Q$ and the rows of $E$ both all sum to 1, so since our scalars 0.85 and 0.15 sum to 1, the rows of $M$ also sum to 1. This can be verified just by summing the rows. Therefore, $M$ is row stochastic, as noted in our proof sketch. Next, we take $M$ and begin multiplying it against some starting vector. Let $\pi_i$ be our PageRank row vector after $i$ iterations. We start with $\pi_0 = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$. Then we just take $M$ and repeatedly multiply it against $\pi_i$. Using Sage to do the matrix multiplication, after one iteration we get:

$$\pi_1 = \pi_0 M = [0.3563, 0.1083, 0.4271, 0.1083]$$

Looking at what we get after 50 and 100 iterations demonstrates the convergence:

$$\pi_{50} = \pi_0 M^{50} = [0.379734313172, 0.145091388731, 0.330082909363, 0.145091388731]$$

$$\pi_{100} = \pi_0 M^{100} = [0.379734313171, 0.145091388731, 0.330082909364, 0.145091388731]$$
Note how the two vectors are the same until the twelfth digit. \( \pi_{50} \) is certainly accurate enough for us, so we have found our PageRank vector. This vector makes sense because our highest ranked page is linked to by 2 pages with relatively high probabilities.

Below is pseudocode for this approach, the amount of iterations we need to run will be discussed below:

**Algorithm 1 Power Iteration**

```
procedure PowerIteration(Adjacency Matrix A, teleportation probability \( \alpha \), number of iterations \( k \))
    // First create the matrix M
    Initialize \( n \times n \) matrix \( M \)
    for \( i \) from 1 to \( n \) do
        numOutLinks ← number of nonzero entries in the \( i^{th} \) row of \( A \)
        for \( j \) from 1 to \( n \) do
            if \( A_{ij} == 0 \) then
                // dealing with nodes with no links, also called dangling nodes
                \( M_{ij} \leftarrow \alpha/n \)
            else
                // We know \( A_{ij} = 1 \)
                \( M_{ij} \leftarrow (1 - \alpha)/numOutLinks \)
        // Now we create the PageRank vector
        Create length \( n \) vector \( \pi \leftarrow [1/n, 1/n, \ldots, 1/n] \)
    for \( i \) from 1 to \( k \) do
        \( \pi \leftarrow \pi M \)
```

To understand how many iterations we need, it is worth taking a closer look at power iteration for a general matrix. This is built off of Larson’s proof of the convergence of the power method [7]. Suppose we have an \( n \times n \) diagonalizable matrix \( M \). By the spectral theorem, \( M \) has eigenvalues \( \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \). Then, we can find representative eigenvectors of these eigenvalues in order to create a basis, call these vectors \( \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\} \). Then, suppose we have an arbitrary vector \( \vec{x} \). We can express \( \vec{x} \) as a linear combination of our basis elements with real scalars \( \{c_1, c_2, \ldots, c_n\} \):

\[
\vec{x} = c_1 \vec{v}_1 + \cdots + c_n \vec{v}_n
\]

Then, when we multiply by \( M^k \) we can use the fact that \( \{\vec{v}_1, \ldots, \vec{v}_n\} \) are all eigenvectors to get:

\[
\vec{x}M^k = c_1 \vec{v}_1 M^k + c_2 \vec{v}_2 M^k + \cdots + c_n \vec{v}_n M^k
= c_1 \lambda_1^k \vec{v}_1 + c_2 \lambda_2^k \vec{v}_2 + \cdots + c_n \lambda_n^k \vec{v}_n
= \lambda_1^k \left( c_1 \frac{\lambda_1}{\lambda_1^k} \vec{v}_1 + c_2 \frac{\lambda_2}{\lambda_1^k} \vec{v}_2 + \cdots + c_n \frac{\lambda_n}{\lambda_1^k} \vec{v}_n \right).
\]

Our goal is to converge to some multiple of the dominant eigenvector, in our case, the one whose entries all add up to zero. For us \( \lambda_1 = 1 \), so this expression gets closer and closer to \( c_1 \vec{v}_1 \) as \( \lambda_2^k/\lambda_1^k \) gets closer to 0. Since our eigenvalues are ordered by size, \( \lambda_i^k/\lambda_1^k \leq \lambda_2^k/\lambda_1^k \) for
any \( i > 2 \), we know those terms will converge at worst as slowly as \( \lambda_2^k/\lambda_1^k \). Similarly, \( \lambda_2 < 1 \) from what we know about the dominant eigenvector, so we know \( \lambda_2^k \) will converge to 0. To figure out how many iterations of the power method we need, we need to know how small \( \lambda_2 \) is.

This might seem bad at first, since \( \lambda_2 \) is not a variable that depends on the input size. Fortunately, Langville et al. supply a discussion of how we know that \( \lambda_2 \) is small enough for convergence to happen quickly. They argue that \( \lambda_2 \) will be the second eigenvalue of our matrix \( Q \) multiplied by \( 1 - \alpha \), 0.85 in our case, and that \( Q \) will almost always have 1 as a second eigenvalue. Then, \( \lambda_2 \) will probably be \( 1 - \alpha \) no matter what the size of the input is, which is good for our time analysis. See Langville for a more detailed discussion [6]. Then, for some error \( \epsilon \), we will need \( (\log(\epsilon) - \log(n - 1))/\log(1 - \alpha) = \log(1/(1-\alpha))(\frac{n-1}{\epsilon}) \) iterations for \( \lambda_2^k \) to be less than \( \epsilon/n - 1 \), which is the number of iterations we need for our result to converge to \( c_1 \lambda_1^k + \epsilon \). Thus, for a constant \( \alpha \) we need \( O(\log(n/\epsilon)) \) iterations. Furthermore, we can see the relationship between \( \alpha \) and the number of iterations from these calculations. Decreasing \( \alpha \) will decrease the base of the logarithm and result in more iterations.

Since matrix multiplication is a fundamental problem, distributed methods of matrix multiplication have existed for a long time. Employing these methods to perform power iteration gives us a simple way of solving the PageRank problem. One important thing to note is that the required matrices we need to multiply are relatively sparse. This comes from the fact that most pages in the web do not link to each other. If we think about all the hundreds of millions of web pages, it is actually relatively rare that two pages link to each other. Pietracaprina et al. provide a distributed algorithm using map reduce to compute sparse-sparse matrix multiplication [14]. The complexity of sparse matrix multiplication relies on the number of nonzero entries in a matrix, but given a constant number of nonzero entries per row we really only need to multiply each entry in the matrix with a constant number of other entries in the matrix. Then, ignoring details of distributing the algorithm, sparse matrix multiplication is \( O(n) \). Given that the number of iterations for convergence is \( O(\log(n/\epsilon)) \), we end up with an \( O(n\log(n/\epsilon)) \), distributed algorithm for computing PageRank.

3.2. Monte Carlo Methods. While the matrix multiplication approach allows us to utilize linear algebraic methods, it is not necessarily the most natural or efficient way to solve the PageRank problem. Another way to see the problem is as a Monte Carlo problem. Here, we would directly simulate random walks across the Markov Chain, and record how often a specific node was visited compared to the total number of nodes visited. An important concept for this algorithm is convergence. The Monte Carlo method relies on random processes, so we are not guaranteed that our results make any sense until we have done enough iterations of the algorithm. We need to figure out how long an algorithm takes to converge on a correct answer. The main issue for this implementation is, how do we know how many random walks we need to do in order to get close to the actual probability of visiting a node [1]? Avrachenkov et al. presents four different algorithms that build off each other when describing Monte Carlo algorithms for PageRank. They refer to the first as “MC end-point with random start”, and argue that this algorithm requires an \( O(n) \) number of random walks. This is a relatively straightforward MC algorithm that involves simulating random walks
from randomly chosen pages, and then evaluates the PageRank of a node based on the num-
ber of walks that end at that node. A slight variation on this algorithm is to iterate through
the nodes and perform some amount of walks per node instead of simply choosing the nodes
randomly. This can control the variance of the algorithm. The algorithm is further improved
by first considering the entire path taken by a random walk instead of just the end point,
and then terminating the paths at dangling nodes, which is a webpage that does not link to
any other page [1].

Below is a simple example for the basic “MC end-point with random start” method, with
the modification that instead of starting at a random node, we simulate \( m \) walks per node
[1]. Let \( G \) be the web graph depicted below, which is the same as the example from before.
First, we need to make some modifications to the graph, similar to the modifications we
made to the matrix in the power iteration example. We create transition probabilities such
that all the transitions from a node add up to 1, and for dangling nodes we act as though
there is a link to every other node in the graph, as shown below. In this case, there are no
dangling nodes, so this modification is not necessary.

![Figure 2. Our web graph.](image1)

![Figure 3. Web graph with transition probabilities.](image2)

Then, all we do is simulate \( m \) random walks starting at each node in the graph, and record
where each random walk ends. To simulate a random walk, we select a random node to start
Then, for each iteration the random walk has an $\alpha$ chance to terminate, where $\alpha$ is our chance for a web surfer to jump to a random page. Again, we use 0.15 for $\alpha$. If the walk does not terminate, we randomly select the next node to move to based on the transition probabilities of the starting node. Repeat this process until the random walk terminates, and then record the node where the walk ended. After simulating $mn$ random walks, we return the PageRank of a node as the number of random walks ending at that node divided by $mn$ [1]. Here is pseudocode for this random walk algorithm:

**Algorithm 2 Random Walks**

```plaintext
// Function to do a walk centered at a node in G
procedure RandomWalk(Graph G, termination chance $\alpha$, start node $u$)
    if a random number in $[0, 1]$ is less than $\alpha$ then
        return $u$
    else
        Randomly select a node $v$ such that there is an edge $(u, v)$ in $G$
        return RandomWalk($v$)

// Function to do $m$ walks per node in $G$ and track the results
procedure WalksPerNode(Graph G, termination chance $\alpha$, integer $m$)
    Initialize an $n$ length vector endCounts
    for node $u \in G$ do
        for $i$ from 1 to $m$ do
            // Do a random walk and track where it terminated
            increment endCounts[RandomWalk($u$)] by 1
    return $\pi = \frac{1}{mn} endCounts$
```

Below are the PageRank vector values after simulating 50 and 100 random walks per node:

\[
\pi_{50} = [0.17, 0.1, 0.61, 0.12] \\
\pi_{100} = [0.3675, 0.1225, 0.35, 0.16]
\]

These are close to the values obtained by the power method, but clearly not as accurate given the number of iterations. However, as mentioned above, there are a number of alterations that Avrachenkov investigates in order to give quicker convergence, such as taking into account all the steps of the random walk, not just the end point [1].

Since these algorithms rely on randomness and approximation, proving that they are correct requires showing that they converge within a certain amount of an answer within a reasonable amount of time. Avrachenkov et al. perform convergence analysis that shows the version of their algorithm which terminates at dangling nodes is the most efficient, at least in the case where there are many dangling nodes in the data. They show that the number of required random walks is $O(n)$ [1]. Since each random walk takes an amount of time depending only on $\alpha$, the time for one random walk is effectively constant. The chance that a random walk does not terminate after $k$ steps is $(\alpha)^k$. Then the average amount of steps a random walk will take before terminating is $\frac{\log(0.5)}{\log(1-\alpha)}$. For a very small $\alpha$ this value will become large, but at $\alpha = 0.15$ this is 4.265. Therefore, this gives us an $O(n)$ PageRank algorithm. The reality of the web is that there are many dangling nodes, so the version that terminates at dangling nodes performs better in the real world case, at least according to their data [1].
course, PageRank can be used in other contexts, since it is really just a graph centralization algorithm, so this algorithm may not necessarily be the best when looking at different graphs, such as a social network instead of a web graph.

When it comes to distributing this version, there are a few different approaches. Avrachenkov suggests having each machine in the cluster perform random walks in parallel [1]. However, it seems that this would require storing the entire graph in the memory of each machine, which may not always be viable for sufficiently large graphs. If the graph had to be stored to disk then this could add to the time complexity and could be space inefficient. Alternatively, the Pregel framework relies on distributing the entire graph across many different machines, in this case information on different nodes in the graph may be spread across different nodes in the cluster [10].

Avrachenkov et al. do not provide a comparison of the speed of Monte Carlo algorithms vs. power iteration methods, but they do compare the relative error across the same number of iterations. Their results were not strictly conclusive. Although the relative error of Monte Carlo methods was significantly lower than power iteration methods after one iteration, upon subsequent iterations the randomness of the Monte Carlo methods meant that relative error did not decrease as reliably as the power iteration method. In fact, sometimes the relative error increased between iterations for a specific web page [1]. Still, Avrachenkov specifies that for important web pages the Monte Carlo method is significantly more accurate. A comparison of strict speed could be valuable, since Avrachenkov et al. only discuss the theoretical amount of time it takes for one iteration of their Monte Carlo methods [1]. You et al. discuss running the PageRank algorithm with an updating data set [20]. In this case, if you want the PageRank vector to be reliable at any time you check it, it might make more sense to use Monte Carlo methods. This is because if it only takes one iteration to get a reasonable PageRank value for a node, then your PageRank vector will still be reasonably good even right after adding new nodes to the dataset. Monte Carlo methods are a powerful option when running only a few iteration of PageRank, but it is not completely clear if they are overall faster or slower than power iteration methods.

An important aspect of both power iteration and Monte Carlo methods, is that they both capture the recursive nature of PageRank. Both of these methods result in a ranking where pages that are linked to by nodes with high ranking also have a high PageRank. This was part of the original plan for PageRank, which bypasses some of the problems associated with simply counting the number of incoming links for a page [13]. After all, links from pages that nobody ever visits are not nearly as meaningful. The power iteration method captures this through the way matrix multiplication works, while the Monte Carlo method directly captures this by using random walks, since if a web surfer ends up at a page, they will be more likely to click on links on that page, and therefore end up at connected pages.

3.3. Distributed PageRank with Link Matrices. Another method involves reformulating PageRank to use link matrices for each individual web page. The goal of this formulation is to come up with an end result that only needs local information, not global information on the whole graph. In other words, if a node is stored on a specific machine, that machine should only need information about neighboring nodes in order to contribute to the PageRank calculation. This way, the number of other machines that need to communicate is minimized, which is essential for minimizing time complexity with a distributed algorithm.
You et al. even show that their method can work with graphs where the size of the network is not known beforehand, and also if new edges are added to the graph over time [20].

This method builds off of work by Ishii et al., who created a distributed algorithm that takes advantage of locality [5]. The number of pages linked to by a single web page will always be relatively small compared to the number of total pages in the web. Having an algorithm where nodes in the graph only need to query nodes they border with is therefore a significant advantage. Although Ishii’s algorithm is randomized, it is different from the Monte Carlo algorithms seen previously. It creates a new matrix for each node, randomly selects matrices to multiply together, and does this for enough iterations such that the result gives the same as result as the power method [5].

These matrices for each node, called link matrices, record only the edges into and out of that specific node. They also have nonzero values along the diagonal of the matrix that preserve the row stochasticity of the matrix, that is they are chosen so that the rows still add up to 1 [5]. Then, matrices are randomly chosen and multiplied together. This strategy may require a lot of memory in order to store each matrix for each node, but there should be a way to restrict this by taking advantage of the fact that the matrices are very sparse. They can only have nonzero values along the diagonal and in the row and column corresponding to the node in the graph, so they are mostly zeroes [5]. Therefore, each matrix might not take up that much memory since we only have to track the nonzero values.

Ishii et al. prove convergence of their algorithm by showing that the result of their algorithm converges to the PageRank vector in the sense of least means squared. In other words, the resulting vector converges to the PageRank vector with respect to the Euclidean Norm [5]. They do this by demonstrating that the average matrix created by this process will converge to the same matrix created as part of the power iteration method. However, they do not seem to give a clear theoretical time bound for their algorithm.

As an example of what these matrices might look like, here are the link matrices corresponding to pages 1 and 2 in the web graph used earlier.

\[
A_1 = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

For \( A_1 \), the first row and column are the same as the columns for the hyperlink matrix. For \( A_2 \) the second row and column match the hyperlink matrix. The row represents the links coming out of the node while the columns represent links going in. The diagonal values are chosen such that the matrices are row stochastic.

Despite the different approaches of these algorithms, they all have the same goal: compute the PageRank of nodes in a graph in a short amount of time. Although a number of different experiments to compare the efficiency of these algorithms have been performed, there does not seem to be any clear consensus on which algorithms win when it comes to pure speed. Additionally, many papers that present a new algorithm seem to approach the problem from the angle of proving that their new algorithm is somehow useful. An unbiased comparison of the speed of each algorithm could, therefore, be a valuable contribution to the literature.
4. PROPOSED METHODS FOR STUDYING THE ALGORITHMS

Aside from discussing the theoretical attributes of the algorithms, I would also like to study the algorithms empirically in order to ascertain the relative strengths and weaknesses of each algorithm. This section contains the methods originally proposed for studying the algorithms, for the actual experiments carried out see Section 6, which differ in the platform used and reduced scope of the experiments. The planned experiments will pay special attention to the process it takes to parallelize each algorithm, and the speed it takes to run each algorithm on a dataset. This would include testing across datasets of different sizes in order to test scalability, since as Washburn notes in his senior thesis from last year, this is not as well covered an area as it should be [18]. In order to keep as many variables under control as possible, the algorithms should all be run on as close to the same platform as possible. Also, due to the popularity of cloud computing for running distributed algorithms, it makes the most sense to run the algorithms in a cloud computing environment. This gives us the ability to run a cluster without managing the servers on our own.

The platform of choice is Amazon Web Services’s EMR, which stands for Elastic MapReduce. This gives us access to a Hadoop cluster in the cloud that can be used to execute distributed algorithms. Although this might restrict us to using MapReduce or Spark, there are libraries for both that work with iterative graph algorithms. These libraries include Apache Giraph and GraphX. Therefore, the main steps would be to:

1. Setup a hadoop cluster using AWS EMR.
2. Install Apache Giraph on the cluster.
3. Implement the following algorithms in Apache Giraph / MapReduce:
   a. Avrachenkov Monte Carlo method terminating at dangling nodes
   b. Power iteration
   c. Ishii’s distributed link matrices method
4. Compare how naturally the approaches lend themselves to the distributed implementation.
5. Run the algorithms on the 855,000 node data set that was released as part of a Google programming competition [9]. A larger dataset may also be necessary.
6. Compare the results across the various data sets in order to understand the scalability and speed of the different algorithms.

A method for comparing how naturally the approaches can be distributed is not immediately obvious. Most likely I will have to use qualitative analysis for this part instead of quantitative analysis. Comparing speed for these algorithms should be more natural, although depending on the consistency of AWS, they could vary considerably for reasons outside the algorithms themselves. Still, in order to keep the research relevant to the world of cloud computing, which is increasingly becoming the future of distributed computing, it is worth restricting this experiment to a cloud computing platform. Ideally, the scalability will agree with the theoretical results in the papers considered. If the results do not agree, I will need to refine my approach, or investigate any possible assumptions or problems with the theoretical results.
By iterating on my approach, I can contribute to the problem of identifying the strongest approach to distributed PageRank.

From the reviewing the literature we can conclude that there are multiple ways to calculate the PageRank vector of a graph, including Monte Carlo methods, power iteration, and distributed link matrices. Current research implies that Monte Carlo methods and power iteration behave similarly, but that Monte Carlo methods are more reliable after the first iteration and may converge more slowly [1]. Further investigation into how these algorithms behave on large datasets using cloud computing could help conclusively determine which implementation is more powerful and scalable. Using modern distributed systems methods in the cloud, my goal is to examine the pros and cons of each algorithm and answer this question.

5. Implementing the Algorithms

The next step is to perform original research and test the algorithms on real world data. In order to actually implement the algorithms we need some kind of framework for doing distributed computation in parallel across multiple machines. Although there are more lightweight frameworks available, I chose to go with a framework that could be easily implemented on a cloud computing platform. Spark was a good choice for these algorithms because of its integrated distributed graph library GraphX, which allows users to easily do large scale graph computations. The code used for this thesis is in the appendix, and can also be viewed on GitHub [19].

Spark is a framework inspired by MapReduce that is designed to do computations on large datasets. The core of Spark consists of two types of functions that can be applied to distributed data sets. The two types are transformations, which turn each piece of data into another piece of data, and therefore do not require moving the data across machines, and actions, which create a piece of data from many pieces of data [15]. Transformations allow high degrees of parallelization, since each piece of data is independent, but actions take more time. Note that the actual power of the platform is not necessarily a factor when comparing two algorithms as long as the algorithms use the same platform, although some sources do say that Spark is faster than MapReduce [16].

5.1. Implementation Details for Power Iteration. As discussed above, the backbone of power iteration is matrix multiplication. Therefore, implementing a way to multiply matrices with vectors and vectors with scalars is enough to create the power iteration algorithm. Remember that in this context we are multiplying a row vector, representing the current PageRank, with a matrix on the right of that vector, representing the hyperlinks between web pages. I based some of my algorithm on an example designed to show people how to use Spark that implements PageRank [17]. This example does not account for dangling nodes, so I had to alter the algorithm accordingly.

In Spark the main data structure is the resilient distributed dataset, or RDD [15]. An RDD stores data in partitions which could be located across different machines in the cluster. If we need to use data from different partitions in order to perform a calculation, for example adding up elements in a matrix after multiplying them with elements in a vector, then we need to shuffle data across partitions. In general, performing operations like map or filter...
where we only have to consider one element at a time do not require a shuffle. However
an operation like reduce or reducing by key requires a shuffle since we need to combine
multiple pieces of data together. Shuffling can be quite expensive in Spark, so creating a
good algorithm often involves minimizing the number of shuffles [21].

This has many similarities to the concept of locality that we discussed earlier. Ideally, the
algorithm would only require combining data within the same partition, so that cross machine
communication is minimized and parallelization is maximized. Of course, it is impossible to
avoid some shuffling, but still the key to an efficient matrix multiplication algorithm requires
keeping data that will have to be used together close together.

Spark gives special significance to RDDs where the entries are tuples with two elements. In
this case there are many operations that treat the first element of the tuple as a key, and
the second element as a value. This is very important for us because multiplying a matrix
and a vector require matching entries in the matrix with corresponding entries in the vector,
multiplying them, and then summing these products appropriately.

In my implementation I store matrices in a simple way. I store matrices as an RDD of entries
[4]. Each entry is just a tuple where the first element is a row and the second is another
tuple that stores the column and value. This allows us to use the row as a key, while still
retaining column and value information. As shown in the diagram, I match vector values
to their corresponding rows using a join. Then I map the joined values by multiplying the
vector and matrix values. Finally, I sum up all values in the same column using reduce by
key. The end result is a new row vector equal to the product of the vector and the matrix.
In the algorithm we use this operation according to the following equation found in [6]. If $H$
is the hyperlink matrix, $\alpha$ is the teleportation factor, $d$ is a column vector of dangling nodes,$n$ is the number of nodes, $e$ is a uniform row vector of ones, and $\pi_i$ is the $i^{th}$ iteration of the
PageRank vector, then we can use
\[
\pi_{i+1} = (1 - \alpha)\pi_i H + ((1 - \alpha)\pi_i d + \alpha)e/n.
\]

The most expensive part of this operation is the matrix multiplication, since the matrix is
significantly larger than any of the vectors. Adding two vectors together requires a join and
a map, but the vectors are smaller than the matrices so this is not as expensive. Multiplying
a vector by a scalar only requires a map, so this operation is relatively inexpensive.

One way to improve data locality is to prepartition the data [21]. Prepartitioning is when
we tell Spark to group certain elements in an RDD together in certain partitions. In my
case I prepartition the data so that entries in the matrix with similar rows are stored close
together. This improves the speed of matching those rows with corresponding vector en-
tries when necessary, since there is a lower chance that spark will have to shift data across
partitions.

5.2. Implementation Details for Graph Methods.

5.2.1. Monte Carlo Methods. The Monte Carlo method I implemented is the complete paths
terminating at dangling nodes as detailed in [1]. To implement this in Spark we match each
node with the following: an array of adjacent nodes, a value tracking the current number of
visitors at each node, and a value tracking the total visitors that have visited that node. Then
we can flatmap each node to a list of randomly selected nodes it links to, one for each visitor
Figure 4. Matrix multiplication process. This takes two shuffles. One shuffle for join and one for reduce by key.

at the start node. We include an \( \alpha \times 100 \) percent chance to terminate the walk, meaning we do not choose a next node. This also happens at dangling nodes. Next, we update the number of visitors and total visitors at each node based on these mapped nodes. Since the walk has a chance to terminate, the number of visitors circulating through the graph decreases over time. Once there are no more current visitors the iteration ends.

After doing \( m \) iterations we calculate the PageRank of each node by dividing the number of times each node has been visited by the total number of visits across the whole graph. As mentioned above, [1] proves that this algorithm gives an equivalent result to the power iteration method in their paper. This requires a join in order to match randomly selected nodes with the values tracking the number of visitors. I also need to reduce by key in order to sum up all the new visitors to each node. In total this is two shuffles to do one step of \( n \) random walks, but we are only dealing with RDDs of \( n \) elements.

Since this algorithm relies on the graph structure of the data, I use Spark’s distributed graph library GraphX. Previous research indicates that the GraphX algorithms have better performance than naive matrix multiplication based methods for PageRank [16]. Therefore, it makes sense that the Monte Carlo methods would have better performance than the matrix multiplication methods. However, as of now the algorithm does not really take advantage
of data locality. This could be improved with a partitioner that is more likely to put nodes with an edge between them in the same partition.

Unfortunately, so far the Monte Carlo methods do not seem to be working as they are supposed to. The error from the Monte Carlo methods is very high compared to the other algorithms, even after a large number of iterations. The error for the Monte Carlo methods will be discussed in the errors section. For this reason Monte Carlo methods were not used in the experiments. Instead, a graph based method built in to Spark was used.

5.2.2. Spark’s Built-In PageRank. Spark also has a built-in PageRank algorithm designed to work with a graph object, with some differences from the PageRank algorithms I implemented [17]. Spark is designed to take advantage of performing an operation on an entire data set, so starting a random walk on one node does not necessarily work as well as other methods. Instead the built in algorithm works somewhat similar to the Monte Carlo methods, with no reliance on randomness. In the Monte Carlo algorithms, we have a random walk starting at a node \( u \). If \( u \) has \( n \) edges coming from it, then the chance the random walk takes any particular edge is \( 1/n \). In the built-in algorithm, instead of actually making a decision and selecting one of the nodes, we can push a rank of \( 1/n \times (1 - \alpha) \times r \), where \( r \) is the current rank of \( u \), to each neighboring node. We also reset the rank of \( u \) to be the sum of the rank received from neighbors. Each node is also sent an additional rank of \( \alpha \). This accounts for our teleportation factor. This algorithm does not redistribute rank from dangling nodes. As discussed later, the error between this algorithm and the matrix methods is very low after enough iterations, which implies that the algorithm is redistributing the rank in some way. From examining the source code it appears that the algorithm normalizes the total rank of the nodes in order to account for the rank lost through dangling nodes, but
only after all iterations are complete [17]. Still, the errors are small enough that it is safe to say this algorithm gives close results to the matrix method.

Previous experiments by other researchers compared this GraphX algorithm to the naive example PageRank algorithm mentioned earlier [16]. Their results showed that the GraphX algorithm was about twice as fast as the other example. It could be worth digging more into the optimizations used by this algorithm and see if they are generally applicable outside the context of Spark and GraphX.

6. Experiments

The goal of the experiments is to test the speed of both the power iteration and built-in graph methods. To test the two algorithms I used clusters of machines through Amazon Web Services. The computers used were m4.large sized machines in all cases. I tested the algorithms on a sample dataset of about 870,000 nodes obtained from a Google programming competition[9]. This data is a fraction of the entire web graph. In order to test how the algorithms behaved with different sized clusters I tested clusters of sizes 2, 4, 6, 8, and 10. I stored the dataset in Amazon S3 and read it in as a text file. Output is written to S3 as well. I recorded time taken for each algorithm on each cluster size as well as the actual results to verify that the amount of error is acceptable. I also recorded a variety of error measurements between the two algorithms.

6.1. Ensuring results are correct. Since PageRank relies on convergence, it can be difficult to actually make sure that the different algorithms give the correct results. To make sure that I am actually getting comparable answers from the different algorithms, I looked at the error between the different PageRank vectors. I considered two kinds of distances between the vectors. One is the Euclidean distance between the two vectors. The other is the distance between the two vectors based on the infinity norm, which is the maximum difference between two matching entries in each vector.

Relative errors between the algorithms at 10, 20, 30, 40, and 50 iterations were calculated using both the infinity and Euclidean norms. Even after 10 iterations the results are comparable to 2 decimal places, and after every ten iterations the error seems to decrease by about a factor of 10.

Since our results are vectors, we need to use a norm in order to measure error [2]. The most well known norm is the Euclidean norm, which is just the geometric distance between two vectors. For a vector $x \in \mathbb{R}^n$ the Euclidean norm of $x$ is given by

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$ 

Another useful measurement is the infinity norm. This is simply given by

$$\|x\|_\infty = \max\{x_1, x_2, \ldots, x_n\}.$$ 

Then given two vectors $x, x' \in \mathbb{R}^n$ we can take the Euclidean norm and infinity norm error of the two vectors by taking $\|x - x'\|_2$ and $\|x - x'\|_\infty$. In order to get a more meaningful measurement that demonstrates the significance of the error relative to the result, we can
take the relative error \[2\]. This is given by \[\frac{\|x - x'\|_2}{\|x\|_2} \text{ and } \frac{\|x - x'\|_\infty}{\|x\|_\infty}\]. Below is a table that includes all of these kinds of errors and demonstrates how the error decreases as we increase the number of iterations of each method. This makes sense since more iterations mean a more accurate calculation.

<table>
<thead>
<tr>
<th>Number of Iterations</th>
<th>Infinity Norm Error</th>
<th>Euclidean Norm Error</th>
<th>Relative Infinity Error</th>
<th>Relative Euclidean Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>17.74623459</td>
<td>75.72102603</td>
<td>0.021965477</td>
<td>0.014226031</td>
</tr>
<tr>
<td>20</td>
<td>3.78854835</td>
<td>14.36991818</td>
<td>0.004680961</td>
<td>0.00270829</td>
</tr>
<tr>
<td>30</td>
<td>0.761175224</td>
<td>2.760171369</td>
<td>9.41E-04</td>
<td>5.21E-04</td>
</tr>
<tr>
<td>40</td>
<td>0.150365821</td>
<td>0.53224979</td>
<td>1.86E-04</td>
<td>1.00E-04</td>
</tr>
<tr>
<td>50</td>
<td>0.029489411</td>
<td>0.102649889</td>
<td>3.65E-05</td>
<td>1.94E-05</td>
</tr>
</tbody>
</table>

Figure 6. Errors between the power iteration method and the graph method.

These relative errors mean that the results are about the same up to 2 digits for 10 iterations, and 5 digits for 50 iterations \[2\]. 2 places of accuracy is not very good, but good enough to say that the results are at least somewhat close, while 5 places of accuracy is strong. Either way, this is enough to say that the algorithms are actually computing the same thing and it is worth comparing them. It would be extraordinarily unlikely for the errors to be this low if the results of the algorithms were not at least extremely similar.

The Monte Carlo methods are not included in this table. For ten iterations the Euclidean error between the Monte Carlo methods and matrix methods was 5322.70 and the relative Euclidean error was approximately 1, which is extremely bad. This is bad enough to say that the Monte Carlo methods and matrix methods are not computing the same thing at all. Even for the small example of four nodes discussed earlier, the Monte Carlo methods give error that is too large. Even after fifty iterations on a small four node example the Monte Carlo methods had a relative error of 0.72 with the matrix methods, which means that the results were not really comparable. On the other hand the matrix and graph methods gave almost exactly the same results after 50 iterations. The results were the same up to 15 decimal places. Most likely this large error was due to incorrect implementation of the algorithm in Spark, and not incorrectness of the algorithm. Overall, this means that the Monte Carlo methods in their current state should not be timed against the other methods. Therefore, I only compared the power iteration methods and the graph methods built in to GraphX.

6.2. Procedure. The procedure for the experiments is as follows.

1. Upload the data sets, both the small web graph from Google \[9\], along with the JAR containing my code to Amazon S3.
2. Run power iteration and the built in GraphX algorithm on EMR with a 2 machine cluster using m4.large size.
3. Set the cluster to terminate after running the algorithms.
4. Record the time taken to run each algorithm and the PageRank vectors outputted by each algorithm.
5. Verify that the errors between algorithms are within acceptable bounds.
(6) Repeat steps 2-5 to run a second trial
(7) Repeat steps 2-6 with clusters of size 4, 6, 8, and 10.

7. Results

Both the matrix multiplication method and the built in graph methods were tested on the cloud using AWS with the following results. The Monte Carlo methods were not included because of huge errors when testing those methods. Below are the results for these experiments.

<table>
<thead>
<tr>
<th>Number of Machines in Cluster</th>
<th>Average of Matrix Time</th>
<th>Average of Built-In Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>349.13</td>
<td>36.53</td>
</tr>
<tr>
<td>4</td>
<td>206.23</td>
<td>24.07</td>
</tr>
<tr>
<td>6</td>
<td>148.02</td>
<td>14.70</td>
</tr>
<tr>
<td>8</td>
<td>269.98</td>
<td>16.68</td>
</tr>
<tr>
<td>10</td>
<td>119.66</td>
<td>10.01</td>
</tr>
</tbody>
</table>

**Figure 7.** Times for different PageRank methods in seconds. Based on two trials.

![Graph of data](image)

**Figure 8.** Graph of data.

First of all, it is clear that the built in graph method is much faster than my matrix multiplication method, by about a factor of 10. However, we can still compare how the two
algorithms behave with respect to an increase in the size of the cluster. Overall, it seems that both algorithms improve with the addition of new machines, although the results are somewhat unexpected when we increase the size of the cluster from 6 to 8 and then 8 to 10 machines. First, I expected the time it takes to calculate to keep going down as we added machines, so the decrease in speed from 6 to 8 machines was unexpected. Additionally, I partitioned the data into 8 pieces when running my algorithm, so I would have expected the speed decrease to occur after 8 machines, not after 6. Furthermore, the speed went back to increasing after adding two more machines, to get to 10 machines.

This fluctuation probably has to do with the tradeoff between high parallelization of the computation and increased amount of inter-network communication required. As the number of machines increases, so does the chance of needing to communicate between two machines during a shuffle. However, it is surprising that this did not become more of a problem when increasing to 10 computers in the cluster.

7.1. Analysis of Results. Overall the two algorithms behaved similarly, although the built-in algorithm did improve more as the number of machines in the cluster increased. On average the matrix method’s speed improved by 65.73% from 2 to 10 machines, while the graph method improved by 72.60%. This gives the graph method a slight edge of 6.87%. It is hard to say if this is statistically significant given the sample size of two trials for each cluster and algorithm. From 4 to 10 machines the matrix method improved by 41.98%, while the graph method improved by 58.41%. This is a greater advantage of 16.43% for the graph method. Overall, it seems that the built-in Spark method utilizing GraphX scales up faster than the matrix multiplication method.

It is worth noting that the built-in method was significantly faster overall than my method. At 2 machines it was 9.56 times faster, and at 10 machines it was 11.95 times faster. There is a possibility that there are optimizations that could be applied to my method that would narrow this gap in overall speed. This would require further research of the GraphX library and how it attains increased efficiency.

As mentioned earlier, another interesting part of this data is the increase in time for both algorithms from 6 machines to 8 machines. This increase seemed to be much more pronounced for the matrix methods. On average the time of the matrix method increased by 82.40% from 6 to 8 machines, while the time of the graph method increased by only 13.44%. This is probably a result of more cross network communication by the matrix method caused by shuffles. Both algorithms also sped up again when increase from 8 to 10 machines. The time of the matrix method decreased by 55.68%, while the time of the graph method decreased by 66.57%. As discussed earlier, this implies that the relative gains from parallelism increased from 8 to 10 machines when compared to the cost of communication across the network, which is surprising since the relative gains decreased from 6 to 8 machines.

8. Future Directions

The algorithms were only tested on a dataset of size about 870,000, but they should also be tested on larger datasets. This could both confirm the theoretical time complexities discussed earlier and test whether a larger dataset affects how the algorithms scale up with more machines in the cluster.
Also, currently I have functions that do a certain number of iterations. However, it might be better to iterate until the error between iterations is small enough. This has the downside of requiring that we compare two vectors at each step, which adds a significant amount of time because we have to use a reduce operation in order to find the greatest error between entries in the vectors. Additionally, I would like to improve the partitioning in the matrix multiplication method, since I believe this is one of the major factors contributing to its current slowness.

Another important thing to improve next would be to reduce the error in the Monte Carlo methods. At the moment they do not seem to be returning any reasonable answer. Once this improvement has been made then the Monte Carlo methods could be compared with the matrix multiplication and built-in graph methods.

A final interesting thing to study in the future is how different partitioning schemes can change the speed of the algorithms. Partitioning data seems to be a very important part of distributed computing, since the less communication is required between computers in the cluster, the more parallelization of the algorithm is possible. The performance decrease between 6 and 8 machines in the matrix methods is an especially interesting part of the data, and could be related to partitioning. Examining how this relates to the tradeoff between increased parallelization and the costs of communication across the network in Spark would also be a good project to look into.

9. Conclusion

The PageRank of a vector is a measure of centrality for nodes in a network. Three ways to calculate the PageRank are using power iteration, Monte Carlo methods, and iterative graph methods using Spark. The graph methods built in to spark and the power iteration method demonstrated a small difference in their performance when increasing the number of machines in a cluster. Running on AWS, the graph methods improved more when increasing the number of machines from 2 to 10 when compared to the power iteration methods. Experiments were carried out using Amazon Web Services and run on a subset of the web graph from Google. Improvements in partitioning could change these results, and could increase the performance of the matrix methods overall.

10. Acknowledgements

Thank you to the entire CS department for their support throughout my time at Haverford. Many thanks to John Dougherty for his advice, assistance, and guidance during the entire thesis process. I would also like to thank Steven Lindell for providing feedback both as a second reader and as the organizer of senior seminar.

REFERENCES


Lecture 22 - Google and Markov Chains, Power Method, SVD.


RDD Programming Guide - Spark 2.3.0 Documentation.


Reza Zadeh. Partitioning for PageRank.

11. Appendix A: Code

This contains the Scala code used to run all experiments. This code was used both to run the experiments on AWS and to check the amount of error between resulting vectors using a personal computer. The variable isAWS controls whether the algorithms should be run in the way that works on the AWS machines or on a personal computer.

```scala
import org.apache.spark.graphx.{Edge, Graph, GraphLoader, VertexId}
import org.apache.spark.rdd.RDD
import java.io.{BufferedWriter, File, FileWriter}

/**
 * Main procedure where I run experiments from
 */
object BasicPageRank {
```
def main(args: Array[String]): Unit = {
  val isAws = true
  val conf = if (isAws) new SparkConf().setAppName("BasicPageRank")
    else new SparkConf().setAppName("BasicPageRank").setMaster("local[2]")
  val sc = new SparkContext(conf)
  if (!isAws) {
    sc.setCheckpointDir("checkpoints")
  }
  sc.setLogLevel("WARN")
  val fileName = if (isAws) "s3://thesisgraphs/web-Google.txt" else "web-Google.txt"
  val file = sc.textFile(fileName)
  val numMachines = if (args.nonEmpty) args(0) else "unknown"
  val experiment = s"${numMachines}machines8partsFIX"
  val matrixOutput = if (isAws) s"s3://thesisgraphs/$experiment/matrixOutput" else "matrixOutput50"
  val graphOutput = if (isAws) s"s3://thesisgraphs/$experiment/graphOutput" else "graphOutput50"
  val errorOutput = if (isAws) s"s3://thesisgraphs/$experiment/errors" else "errors10partfix"
  val timesOutput = if (isAws) s"s3://thesisgraphs/$experiment/times" else "times10partfix"
  val numIters = 10
  // val smallPowerPageRanks = matrixMethodTest(sc)
  // graphMethodTest(sc, Some(smallPowerPageRanks))

  // Matrix Method First
  val adjacencyMatrix = MatrixMethod.fileToMatrix(file).partitionBy(new HashPartitioner(8)).persist()
  val startTime = System.nanoTime()
  val powerIterationResult = MatrixMethod.powerIterations(
    adjacencyMatrix, sc, numIters, 0.85)
  val timeTaken = (System.nanoTime() - startTime) / 1e9d
  // powerIterationResult.getValues.coalesce(1).saveAsTextFile(matrixOutput)
  adjacencyMatrix.unpersist()

  // Built in Graph Method
  val webGraph = GraphLoader.edgeListFile(sc, fileName, numEdgePartitions = 8)
  val graphStartTime = System.nanoTime()
  val rankedGraph = webGraph.staticPageRank(numIters).vertices
  // (0.001).vertices
val graphTimeTaken = (System.nanoTime() - graphStartTime) / 1e9d
val graphRanks = new DistrVector(rankedGraph.map {
  case (id, value) => (id.toInt, value)
})

// rankedGraph.coalesce(1).saveAsTextFile(graphOutput)

// Monte Carlo Method
// val mcPageRanks = new DistrVector(MonteCarloMethod.doMNWalks(
//   webGraph, numIters).map({
//     case (key, value) => (key.toInt, value)
// })))

val infError = powerIterationResult.infNormDistance(graphRanks)
val euclideanError = powerIterationResult.euclidDistance(graphRanks)

val infRelativeToMatrix = infError / powerIterationResult.infNorm()
val eucildRelativeToMatrix = euclideanError / powerIterationResult.euclidNorm()

val times = sc.parallelize(Seq(
  ("Power Iteration", timeTaken),
  ("Built-in", graphTimeTaken)
), 1)
times.saveAsTextFile(timesOutput)
val errors = sc.parallelize(Seq(
  ("Infinity Error", infError),
  ("Euclidean Error", euclideanError),
  ("Infinity Error Relative to Matrix", infRelativeToMatrix),
  ("Euclidean Error Relative to Matrix", eucildRelativeToMatrix),
  ("Infinity Error Relative to Graph", infRelativeToGraph),
  ("Euclidean Error Relative to Graph", eucildRelativeToGraph)
), 1)
errors.saveAsTextFile(errorOutput)

// matrixMethodTest(sc)
def vectorTest(sc: SparkContext): Unit = {
    val testVector = new DistrVector(sc.parallelize(Seq((0, 5.0))))
    val testMatrix = sc.parallelize(Seq((0, (0, 4.0))))
    val multiplied = testVector.matrixMult(testMatrix)
    // multiplied . getValues . foreach (x => println (x))

    val biggerVector = new DistrVector(sc.parallelize(Seq((0, 5.0), (1, 3.0), (2, 2.0))))
    val otherVector = new DistrVector(sc.parallelize(Seq((0, 4.0), (1, 1.0), (2, 4.0))))
    val fatMatrix = sc.parallelize(Seq(
        (0, (0, 2.0)), (0, (1, 3.0)),
        (1, (0, 1.0)), (1, (1, 2.0)),
        (2, (0, 0.0)), (2, (1, 0.0))
    ))
    val bigMultiplied = biggerVector.matrixMult(fatMatrix)
    bigMultiplied.getValues.foreach(x => println(x))
    println(biggerVector.euclidDistance(otherVector))
}

def matrixMethodTest(sc: SparkContext): DistrVector = {
    val sparseMatrix = sc.parallelize(Seq(
        (0, (0, 2.0)), (0, (1, 3.0)),
        (1, (0, 1.0)), (1, (1, 2.0)),
        (2, (0, 0.0)), (2, (1, 0.0))
    ))
    // MatrixMethod . getDanglers ( sparseMatrix , 6, sc). foreach (x => println (x))
    val origExampleAdj = sc.parallelize(Seq(
        (0, (1, 1.0)), (0, (2, 1.0)), (0, (3, 1.0)),
        (1, (2, 1.0)),
        (2, (0, 1.0)), (3, (0, 1.0)), (3, (2, 1.0))
    ))
    val withDanglers = sc.parallelize(Seq(
        (0, (1, 1.0)), (0, (2, 1.0)), (0, (3, 1.0)),
        (1, (2, 1.0)),
        (3, (0, 1.0)), (3, (2, 1.0))
    ))
    val twoDanglers = sc.parallelize(Seq(
        (0, (1, 1.0)), (0, (2, 1.0)), (0, (3, 1.0)),
        (1, (2, 1.0)), (1, (4, 1.0)),
        (3, (0, 1.0)), (3, (2, 1.0))
    ))
    // val hyperlinks = MatrixMethod . toHyperLinkMat ( origExampleAdj )
    // val danglers = MatrixMethod . getDanglers ( origExampleAdj , numNodes , sc)
    // val uniform = new DistrVector ( sc . parallelize ( Seq ((0, 0.25), (1, 0.25), (2, 0.25), (3, 0.25))))
```scala
// MatrixMethod.iterate(uniform, hyperlinks, danglers, 0.85, 4, sc).
println()
val startTime = System.nanoTime()
val powerIterationResult = MatrixMethod.powerIterations(withDanglers, sc, 50, 0.85)
val timeTaken = (System.nanoTime() - startTime) / 1e9d
val twoDanglerResult = MatrixMethod.powerIterations(twoDanglers, sc, 50, 0.85)
println(timeTaken)
println("PRINTING POWER METHOD WITH ONE Dangler")
powerIterationResult.printAll()
println("DONE WITH POWER METHOD")
println("PRINTING POWER METHOD WITH TWO Danglers")
twoDanglerResult.printAll()
println("DONE WITH TWO Dangler POWER METHOD")

/*
val joinTester1 = sc.parallelize(Seq((0, 1), (1, 2))
val joinTester2 = sc.parallelize(Seq((1, 3)))
joinTester1.join(joinTester2).foreach(x => println(x))
*/

def graphMethodTest(sc: SparkContext, errorAgainst: Option[DistrVector] = None): Unit = {
  val vertices: RDD[(VertexId, Int)] = sc.parallelize(Array((0L, 1), (1L, 1), (2L, 1), (3L, 1)))
  val edges = sc.parallelize(Array(
    Edge(0L, 1L, 1), Edge(0L, 2L, 1), Edge(0L, 3L, 1),
    Edge(1L, 2L, 1),
    Edge(3L, 0L, 1), Edge(3L, 2L, 1)
  ))
  val danglerGraph = Graph(vertices, edges)
  val startTime = System.nanoTime()
  val pageRanks = danglerGraph.staticPageRank(50).vertices
  val timeTaken = (System.nanoTime() - startTime) / 1e9d
  println("PRINTING GRAPH TEST WITH Dangler")
  pageRanks.foreach(println)
  println("DONE WITH GRAPH TEST")
  val twoDanglerVers: RDD[(VertexId, Int)] = sc.parallelize(Array((0L, 1), (1L, 1), (2L, 1), (3L, 1), (4L, 1)))
  val twoDanglerEdge = sc.parallelize(Array(
    Edge(0L, 1L, 1), Edge(0L, 2L, 1), Edge(0L, 3L, 1),
    Edge(1L, 2L, 1), Edge(1L, 4L, 1),
    Edge(3L, 0L, 1), Edge(3L, 2L, 1)
  ))
  val twoDanglerGraph = Graph(twoDanglerVers, twoDanglerEdge)
  val twoDanglerRanks = twoDanglerGraph.staticPageRank(50).vertices
  println("PRINTING GRAPH TEST WITH TWO Danglers")
```

Code to create a vector object to use with Spark:

```scala
import org.apache.spark.mllib.linalg.distributed.CoordinateMatrix
import org.apache.spark.rdd.RDD

// Vector object that contains useful methods for use on an RDD such as
// Matrix multiplication and error calculation.

class DistrVector(values: RDD[(Int, Double)]) {
  def getValues: RDD[(Int, Double)] = values

  def scale(scalar: Double): DistrVector = {
    new DistrVector(values.mapValues { value => scalar * value })
  }

  // assuming matrices of the form (row, (col, value)) and the vector is
  // on the left, so that indices of the vector
  // match to rows
  def matrixMult(mat: RDD[(Int, (Int, Double))]): DistrVector = {
```

val withScalars = values.join(mat)
// We just care about the column and vector value * matrix value
val newValues = withScalars map {
  case (_, (vecValue, (col, matValue))) => (col, vecValue * matValue)
} reduceByKey {
  case (x, y) => x + y
}

new DistrVector(newValues)

def euclidDistance(other: DistrVector): Double = {
  // Find differences squared, then reduce
  Math.sqrt(values.fullOuterJoin(other.getValues).map {
    case (_, (x1, x2)) => Math.pow(x1.getOrElse(0.0) - x2.getOrElse(0.0), 2)
  }).reduce {
    (a, b) => a + b
  })
}

def infNormDistance(other: DistrVector): Double = {
  values.fullOuterJoin(other.getValues).map {
    case (_, (x1, x2)) => Math.abs(x1.getOrElse(0.0) - x2.getOrElse(0.0))
  }.reduce {
    (a, b) => Math.max(a, b)
  }
}

def infNorm(): Double = {
  values.map(_._2).reduce(Math.max)
}

def euclidNorm(): Double = {
  Math.sqrt(values.map {
    case (_, x) => Math.pow(x, 2)
  }.reduce(_ + _))
}

def addRDD(rdd: RDD[(Int, Double)]: DistrVector = {
  new DistrVector(values.fullOuterJoin(rdd).mapValues {
    case (vectorVal, otherVal) => vectorVal.getOrElse(0.0) + otherVal.getOrElse(0.0)
  })
}

def cache(): DistrVector = {
  new DistrVector(values.cache())
}
override def toString: String = "DistrVector(" + values.toString() + ")"

// For debugging only, bad performance in parallel maybe
def printAll(): Unit = {
  values.foreach(x => println(x))
}

Code to run power iteration:

```scala
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix
import org.apache.spark.rdd.RDD

/*
  * Power Iteration Method
  */

// https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture16/
// Partitioning_PageRank.pdf
object MatrixMethod {

  def fileToMatrix(file: RDD[String]): RDD[(Int, (Int, Double))] = {
    file.map { line =>
      val edge = line.split("\t")
      (edge(0).toInt, (edge(1).toInt, 1.0))
    }
  }

  def getDanglers(adjacencyMatrix: RDD[(Int, (Int, Double))], numNodes: Long, sc: SparkContext, nodes: RDD[Int]): RDD[(Int, Int)] = {
    val notDanglers = adjacencyMatrix.map {
      case (node, _) => node
    }
    nodes.subtract(notDanglers).map(x => (x, 1))
  }

  def getNodes(adjacencyMatrix: RDD[(Int, (Int, Double))]): RDD[Int] = {
    adjacencyMatrix.flatMap({
      case (row, (col, _)) => Iterable(row, col)
    }).distinct()
  }

  def toHyperLinkMat(adjacencyMatrix: RDD[(Int, (Int, Double))]): RDD[(Int, (Int, Double))] = {
    val linkCounts = adjacencyMatrix.aggregateByKey(0)((accum: Int, x: ((Int, Double))) => accum + 1,
    (accum1: Int, accum2: Int) => accum1 + accum2)
    adjacencyMatrix
  }

```
join(linkCounts)
.map {
  case (row, ((col, _), numLinks)) => (row, (col, 1.0 / numLinks))
}

// Idea: pik+1 = alpha * pik * hyperlinks + (alpha * pik * danglers + 1
= (alpha * pik * uniform / n

def iterate(pivector: DistrVector, hyperlinks: RDD[(Int, (Int, Double))],
danglers: RDD[(Int, Int)], alpha: Double,
umNodes: Long, sc: SparkContext, nodes: RDD[Int]):
  DistrVector = {
    // debug("Number of pageRanks at start of iteration: " + pivector.
    getValues.count())
    val hyperLinkPart = pivector.scale(alpha).matrixMult(hyperlinks)
    // debug("Number of ranks in hyperlink part: " + hyperLinkPart.
    getValues.count())

    val pivectorTimesDanglers = pivector.getValues.join(danglers).fold((0,
      (0,0))) {
      case (((_, (value1, _)), (_, (value2, _)))) => (0,(value1 + value2, 0)
    }
    val danglerPart = nodes.map( index => (index, (alpha *
      pivectorTimesDanglers + 1 - alpha) / numNodes))
    // debug("Number of pageRanks in danglerPart: " + danglerPart.count())
    val result = hyperLinkPart.addRDD(danglerPart)
    // debug("Number of pageRanks at end of iteration: " + result.getValues
    .count())
    result
  }

  def powerIterations(adjacencyMatrix: RDD[(Int, (Int, Double))], sc:
    SparkContext, numIterations: Int, alpha: Double): DistrVector = {
    val nodes = getNodes(adjacencyMatrix).cache()
    val numNodes = nodes.count()
    val danglers = getDanglers(adjacencyMatrix, numNodes, sc, nodes).cache()
    val hyperlinks = toHyperLinkMat(adjacencyMatrix).cache()
    var pivector = new DistrVector(nodes.map(x => (x, 1.0 / numNodes)))
    for (i <- 1 to numIterations) {
      // debug("Starting iteration " + i)
      val nextPivector = iterate(pivector, hyperlinks, danglers, alpha,
        numNodes, sc, nodes).cache()
      pivector = nextPivector
    }
    pivector.scale(numNodes)
  }

  def debug(str: String): Unit = {
val DEBUG = false
if (DEBUG) {
    println(str)
}

// Start of possibly cleaner way of doing things
// Adapted from https://github.com/apache/spark/blob/master/examples/src
// /main/scala/org/apache/spark/examples/SparkPageRank.scala
// Add in dealing with dangling nodes
/*
   def powerUntilConvergence(adjacencyList: RDD[(Long, Iterable[Long])],
                              numNodes: Int,
                              sc: SparkContext, tolerance: Double, alpha: Double = 0.15): RDD[(Long, Double)] = {
      val hyperlinks = adjacencyList.mapValues(outLinks => {
        val size = outLinks.size // cache the size
        outLinks.map(id => (id, 1.0 / size))
      })

      // Create danglers
      val possibleNodes = sc.parallelize(0L until numNodes)
      val notDanglers = adjacencyList.map {
        case (node, _) => node
      }
      possibleNodes.subtract(notDanglers)

      val ranks = possibleNodes.map(n => (n, 0, 1, false)) //(index, old
      val finishedCount = sc.longAccumulator
      while (finishedCount.value != numNodes) {
        val newTraffic = ranks.
      }
    }
*/

Code to run Monte Carlo methods (ultimately not used in the speed comparisons):

import org.apache.spark.graphx.{EdgeDirection, Graph}
import org.apache.spark.rdd.RDD
import scala.util.Random

/*
   * Code for running the Monte Carlo Methods
   */
object MonteCarloMethod {
    val r = new Random()
    def doMNWalks(webGraph: Graph[Int, Int], numIters: Int, alpha: Double = 0.85): RDD[(Long, Double)] = {

val numNodes = webGraph.vertices.count()
val neighbors = webGraph.collectNeighborIds(EdgeDirection.Out)
var walksGraph = webGraph.mapVertices(
    (_, _) => (1, 1, Array[Long]()) // Number of current visitors, total
    number of visitors
).joinVertices(neighbors)({
    case (_, _, neighbs) => (1, 1, neighbs)
})
for (i <- 1 to numIters) {
    // println("Iteration " + i)
    var nextVisits = getRandomVisits(walksGraph, alpha) // Collect all
    visits to each node
    // Reset and add visits
    while (!nextVisits.isEmpty()) { // Don’t terminate until all walks
        have visited dangling nodes
        // println(nextVisits.count())
        walksGraph = walksGraph.outerJoinVertices(nextVisits)({
            case (_, (_, totalVisits, neighbs), maybeVisits) => {
                val newVisits = maybeVisits.getOrElse(0) // Default to 0 new
                visitors
                // println("New visitors " + newVisits)
                // println("Before updating " + totalVisits)
                (newVisits, totalVisits + newVisits, neighbs) // All the
                current visitors have left, add the new visitors to the
                total
            }
        })
        nextVisits = getRandomVisits(walksGraph, alpha)
        walksGraph = walksGraph.mapVertices({ // Reset current visitors to
            1
            case (id, (_, totalVisits, neighbs)) => (1, totalVisits, neighbs
        })
        walksGraph.checkpoint()
    }
}
val combinedVisits = walksGraph.vertices.map(
    case (_, (_, visits, _)) => visits
).reduce(_ + _)
// println(combinedVisits)
walksGraph.vertices.map(
    case (id, (_, visits, _)) => (id, visits.toDouble / combinedVisits)
})
}

def getRandomVisits(walksGraph: Graph[(Int, Int, Array[Long]), Int],
   alpha: Double): RDD[(Long, Int)] = {
   walksGraph.vertices.flatMap(
       case (id, (numSurfers, _, neighbs)) => {
       ...
// Randomly select a next node to visit
if (!neighbs.isEmpty) {
    for (j <- 1 to numSurfers; if r.nextDouble() < alpha) yield (neighbs(r.nextInt(neighbs.length)), 1)
    } else {
    Array[(Long, Int)]()
    }
}).reduceByKey(_ + _)