IE494 Big Data Processing

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar, Gujarat 382007

PageRank Computation

Project Report

By

202101209 Nikita Shah
202101231 Rohan Mistry

Under the guidance of:

Prof. PM Jat

Autumn AY 2024-25

18.11.2024

Abstract:

PageRank is a link analysis algorithm developed by Larry Page and Sergey Brin, originally for ranking web pages in search engine results. It measures the relative importance of nodes in a directed graph by considering the link structure, treating links as votes of importance. The algorithm operates iteratively, assigning each node a rank value based on the ranks of its inbound nodes and the number of outbound links they possess. PageRank is one of the principle criteria according to which Google ranks Web pages. PageRank can be interpreted as a frequency of visiting a Web page by a random surfer and thus it reflects the popularity of a Web page.

PageRank applies the concept of a random surfer, who randomly clicks on links with a probability (d) (damping factor), and teleports to a random node with probability (1-d). This ensures convergence and resilience to disconnected graphs. Widely utilized beyond web search, PageRank finds applications in citation analysis, social networks, recommendation systems, and bioinformatics. Its simplicity, scalability, and theoretical robustness make it a cornerstone of graph theory and network analysis. This paper delves into the core principles, mathematical foundation, practical implementations, and real-world applications of the PageRank algorithm.

Table of Content:

Sr. No	Topics
1	Introduction
2	Google Matrix
3	Power Iteration
4	Monte Carlo Methods
5	Power Iteration vs Monte Carlo
6	PageRank Simulation using Map-Reduce
8	PageRank Simulation using Apache Spark
9	References

Introduction:

PageRank is an algorithm initially developed by Larry Page and Sergey Brin as part of Google's search engine. It is used to rank web pages based on their importance within a network. The algorithm models the behavior of a "random surfer" who navigates through web pages by following hyperlinks, with occasional random jumps to a new page. The PageRank score of a page reflects its significance, determined by the quantity and quality of links pointing to it.

Mathematically, PageRank computes the stationary distribution of a Markov chain defined by the web graph. This stationary distribution gives the probability of landing on a particular page after a large number of random transitions.

Google Matrix:

On the internet, the world wide web, or web, is the collection of all websites accessible by the public from a web browser. Within these websites, also known as webpages, there are clickable links from one page to another. A user surfing the web can start at an arbitrary webpage and click on its hyperlinks to navigate to other webpages.

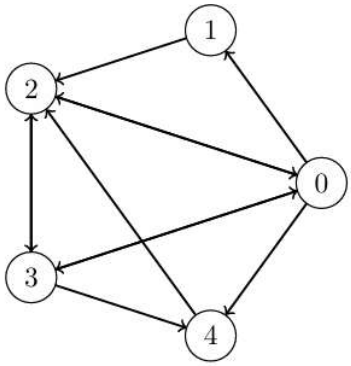


Fig. 1 An example of web graph with five web pages

$$\hat{A} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Link Matrix for directed Graph in Fig. 1

Let A be the link matrix corresponding to the graph G(A)with its nonzero columns normalized, e belongs to Rn be the column vectors with all entries equal to one and let d belongs to Rn be a column vector such that dj=1 if vertex j is a dangling node, or dj=0 otherwise.Then

$$M = (1-m)\left(A + \frac{1}{n}ed^T\right) + \frac{m}{n}ee^T$$

Where 0<m<1. M is the Google Matrix. m=015 is the value initially used by Google.

The Google matrix will always have 1 as simple eigenvalue. Then M will have a unique normalized eigenvector among all positive entries associated to this eigenvalue, resulting in a **unique ranking.**

Power Iteration:

Power iteration is one of the oldest methods for computing the dominant eigenvalue of a matrix and the corresponding eigenvector . The dominant eigenvalue is the eigenvalue with the largest absolute value. Power iteration is the known method used by Google to compute the PageRank eigenvector in practice.

Pseudocode:

Algorithm 1 Power Iteration for Google Matrix

Function PowerIteration(A, x, α, ϵ, max_iters)
  Input: Sparse matrix A (Google matrix), initial vector x, damping factor α, convergence threshold ϵ, maximum iterations max_iters
  Output: Dominant eigenvector v, number of iterations iter_count
$v \leftarrow x$ ▷ Initialize vector with initial probabilities
$iter\_count \leftarrow 0$
while $iter\_count <$ max_iters **do**
    $v_{new} \leftarrow \alpha \cdot A \cdot v + (1 - \alpha) \cdot x$ ▷ Compute the next vector
    $v_{new} \leftarrow \text{Normalize}(v_{new})$ ▷ Normalize the vector
    if $\|v_{new} - v\| < \epsilon$ **then** ▷ Check for convergence
      break ▷ Exit loop if converged
    end if
    $v \leftarrow v_{new}$ ▷ Update the vector for the next iteration
    $iter\_count \leftarrow iter\_count + 1$
end while
Return $v, iter\_count$

Function Normalize(v)
  Input: Vector v
  Output: Normalized vector v
$norm \leftarrow \|v\|$ ▷ Compute the norm of the vector
if $norm \neq 0$ **then**
    $v \leftarrow v/norm$ ▷ Normalize by dividing each element by the norm
end if
Return v

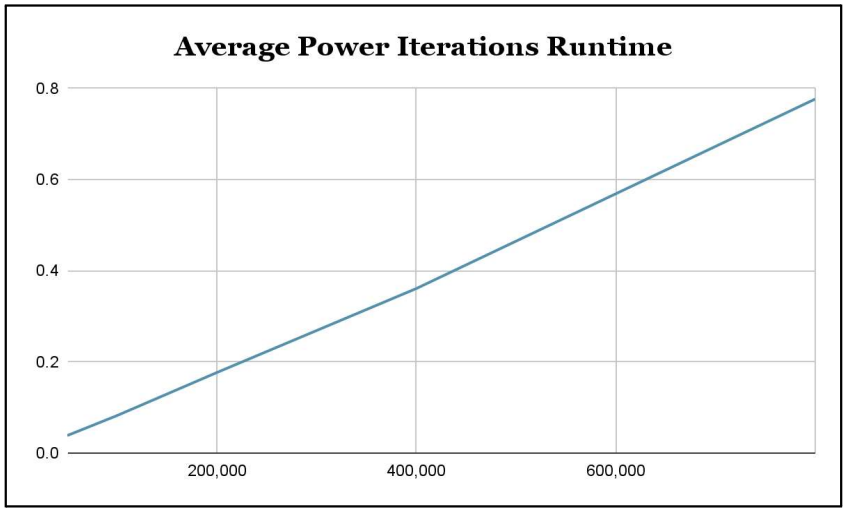
Main Program:
  Read vertex size from command-line argument
  Create a file to log results: file = OpenFile("powertimes_".vertex_size."_.txt")
for $i = 0$ **to** NO_OF_GRAPHS **do**
    Read sparse matrix from file: A = LoadSparseMatrix("sparse_".i."_.csv")
    Initialize the probability vector: x = InitializeVector(vertex_size)
    Start timer
    Perform power iteration: $v, iter\_count \leftarrow$ PowerIteration($A, x, \alpha = 0.15, \epsilon = 1e-8$, max_iters = 100000)
    End timer
    Log the results: time_elapsed, iter_count to file
end for
  Close the output file

Power Iteration Code: Click here

PageRank Outputs: Click here

Results:
We ran the power iteration using m = 0.15 on the web graphs generated to test how the algorithm scales as the graphs double in size. There were 5 graphs of each size. It is easy to observe that the power iteration converges linearly. We can only expect the rate of convergence to be at most 1-m=0.85. In particular, the generated data showed that the estimated rates of convergence never exceeded 0.85 and were much smaller in general, providing a much faster convergence.

Number of vertices	Average Rate of Convergence	Average Runtime (in seconds)
50,000	0.27175	0.03841
100,000	0.26614	0.08232
200,000	0.25634	0.17666
400,000	0.25621	0.36095
800,000	0.24689	0.77673



Advantages:

- **Simplicity**: The algorithm is easy to implement and understand.
- **Efficiency**: Works well for large, sparse matrices since it avoids complex matrix factorizations.
- **Scalability**: Can be applied to very large matrices using distributed computation.

Limitations:

- **Only for Dominant Eigenvalue**: The method finds only the eigenvalue with the largest magnitude.
- **Sensitivity**: May converge slowly if the matrix has eigenvalues of nearly equal magnitude.
- **Non-convergence**: Requires the matrix to have a unique dominant eigenvalue for guaranteed convergence.

Monte Carlo Methods:

Monte Carlo methods for PageRank computation estimate the rank of nodes by simulating random walks on the graph. Starting from a given node, random walks are performed, and the frequency of visits to each node approximates its PageRank. These methods are especially useful for large, sparse graphs due to their simplicity, scalability, and natural parallelism. Monte Carlo methods are a probabilistic approach to compute PageRank by simulating the behavior of random surfers navigating through a graph. Instead of solving the PageRank equation directly, these methods rely on repeated random walks starting from various nodes. The frequency with which nodes are visited during these walks provides an estimate of their PageRank values.

This approach is particularly well-suited for large-scale graphs, as it avoids the computational cost of matrix operations required in traditional iterative methods like power iteration. Monte Carlo methods are also highly parallelizable, with each random walk being independent, making them ideal for distributed or GPU-based systems. Despite their simplicity and scalability, the accuracy of the results depends on the number of random walks performed and the quality of the sampling strategy.

Monte Carlo Code: Click here

Monte Carlo Output Files: Click here

Monte Carlo Method 1: Endpoints With Random Start

This Monte Carlo Method simulates a "random walk" process to compute PageRank through sampling. Random surfers traverse the graph, occasionally "getting bored" and jumping to a random node.

The Monte Carlo approach involves simulating a random walk across the web graph, where the surfer:

- Starts at a random web page.
- Randomly follows an outgoing link with probability d (damping factor).
- Teleports to a random web page with probability $1-d$.

By tracking the frequency with which each page is visited during these simulations, an approximate PageRank value can be calculated for each page.

Steps:

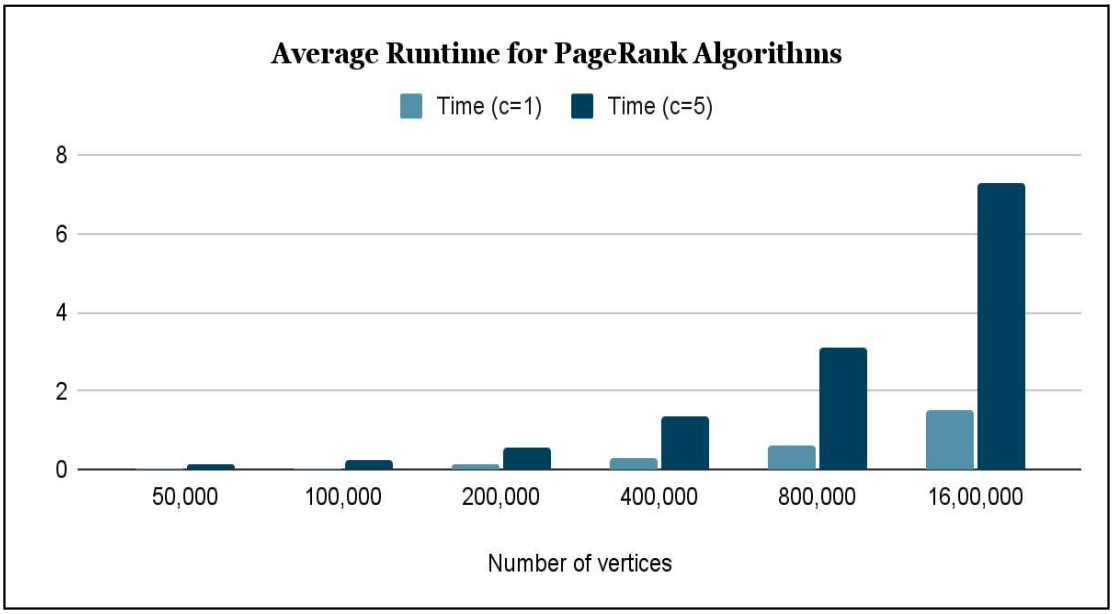
- Random Walks: Perform multiple random walks starting from randomly selected nodes.
- Track Visits: Maintain a counter for each node to record the number of visits during all random walks.
- Normalize Results: Normalize the counters by the total number of steps taken across all walks to estimate the relative PageRank values. Normalize so that the sum of all entries equals 1, turning it into a probability distribution (valid PageRank vector).

PseudoCode:

Algorithm 2 Monte Carlo Method 1 for PageRank Estimation

Function MonteCarloPageRank(N, m)
  Input: N (number of Monte Carlo simulations), m (damping factor)
  Output: Approximate PageRank vector x_{mc}
Initialize $x_{mc} \leftarrow$ zeros($num\_vertices$) ▷ Initialize the PageRank vector
$counter \leftarrow \frac{1}{N}$ ▷ Counter for updating the PageRank value
for $i = 0$ **to** $N - 1$ **do**
  $w \leftarrow$ UniformRandom$(0, num_vertices)$ ▷ Choose a random start node
  while True **do**
    $bored \leftarrow$ UniformRandom$(0, 1)$ ▷ Randomly decide if the walk ends
    if $bored < m$ **then**
      break ▷ End walk if surfer gets bored
    end if
    $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of node w
    if $numNeighbors = 0$ **then**
      $w \leftarrow$ UniformRandom$(0, num_vertices)$ ▷ Randomly jump if dangling node
    else
      $offset \leftarrow$ UniformRandom$(0, numNeighbors)$ ▷ Pick a random neighbor
      $w \leftarrow r[p[w] + offset]$ ▷ Move to a random neighbor
    end if
  end while
  $x_{mc}[w] \leftarrow x_{mc}[w] + counter$ ▷ Increment PageRank estimate for node w
end for
$sum \leftarrow \sum_{i=0}^{num\_vertices-1} x_{mc}[i]$ ▷ Sum all values in x_{mc}
if $sum \neq 0$ **then**
  for $i = 0$ **to** $num\_vertices - 1$ **do**
    $x_{mc}[i] \leftarrow \frac{x_{mc}[i]}{sum}$ ▷ Normalize the PageRank vector
  end for
end if
Output: x_{mc} ▷ Final PageRank vector

Number of vertices	Time (c=1)	Time (c=5)
50,000	0.0248	0.1221
100,000	0.0515	0.2597
200,000	0.1127	0.5573
400,000	0.2778	1.3792
800,000	0.6115	3.1077
16,00,000	1.5095	7.2822



Monte Carlo Method 2: Endpoints with Cyclic Start

Inputs:

- q: Number of random walks starting from each vertex.
- m: Damping factor, representing the probability that a random walk terminates at any step (0.15).

- counter: The contribution to the PageRank value for each visit during the random walks.

Monte Carlo Simulation:

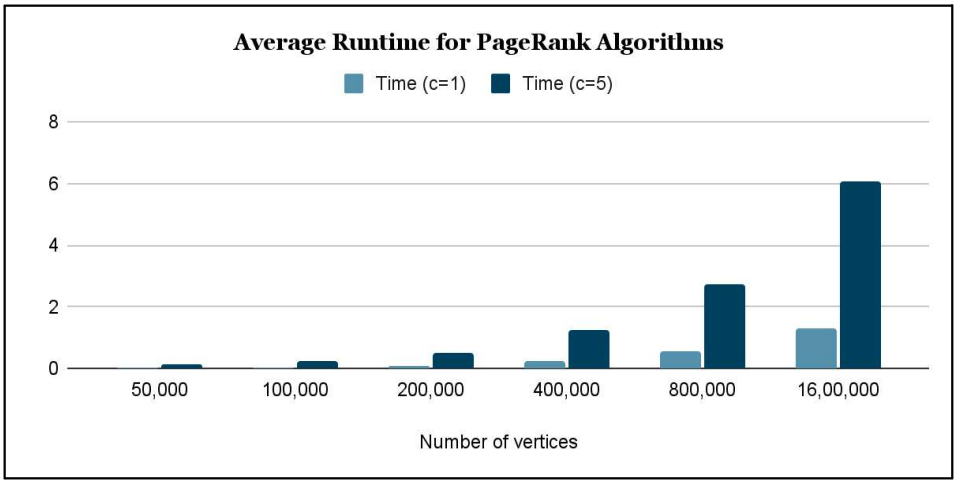
- For each vertex:
  - Perform q random walks starting from the vertex.
  - During each walk, decide whether to terminate based on the boredom probability (bored).
  - If the vertex is a dangling node (no outgoing links), randomly jump to any node.
  - Otherwise, randomly choose one of its neighbors and continue the walk.
  - Increment the PageRank value of the ending node of the walk by the counter.
  -

PseudoCode:

Algorithm 3 Monte Carlo Method (mc2) for PageRank Estimation

Function MonteCarloPageRank(q, m)
  Input: q (number of random walks per vertex), m (damping factor)
  Output: Approximate PageRank vector x_{mc}
Initialize $x_{mc} \leftarrow \text{zeros}(num\_vertices)$ ▷ Reset PageRank vector
$counter \leftarrow \frac{1}{q \cdot num\_vertices}$ ▷ Contribution per visit
for $i = 0$ **to** $num\_vertices - 1$ **do**
  for $dummy = 0$ **to** $q - 1$ **do**
    $w \leftarrow i$ ▷ Start the random walk at vertex i
    while True **do**
      $bored \leftarrow \text{UniformRandom}(0, 1)$ ▷ Check if surfer gets bored
      if $bored < m$ **then**
        break ▷ End walk if bored
      end if
      $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of vertex w
      if $numNeighbors = 0$ **then**
        $w \leftarrow \text{UniformRandom}(0, num\_vertices)$ ▷ Jump to a random vertex
      else
        $offset \leftarrow \text{UniformRandom}(0, numNeighbors)$ ▷ Choose random neighbor
        $w \leftarrow r[p[w] + offset]$ ▷ Move to the neighbor
      end if
    end while
    $x_{mc}[w] \leftarrow x_{mc}[w] + counter$ ▷ Increment count for ending vertex
  end for
end for
$sum \leftarrow \sum_{i=0}^{num\_vertices-1} x_{mc}[i]$ ▷ Sum all elements in x_{mc}
if $sum \neq 0$ **then**
  for $i = 0$ **to** $num\_vertices - 1$ **do**
    $x_{mc}[i] \leftarrow \frac{x_{mc}[i]}{sum}$ ▷ Normalize to get probabilities
  end for
end if
Output: x_{mc} ▷ Final PageRank vector

Number of vertices	Time (c=1)	Time (c=5)
50,000	0.0235	0.1191
100,000	0.0488	0.2491
200,000	0.1054	0.5255
400,000	0.2538	1.2376
800,000	0.5501	2.7091
16,00,000	1.2969	6.0964



Monte Carlo Method 3: Complete Path

- q: Number of random walks initiated from each vertex.
- m: Damping factor, representing the probability that a random walk terminates at any step (0.15).

- counter: The contribution to the PageRank value for each visit (fixed at 1.0).
- total_visits: Tracks the total number of visits across all random walks.

Monte Carlo Simulation:

For each vertex:

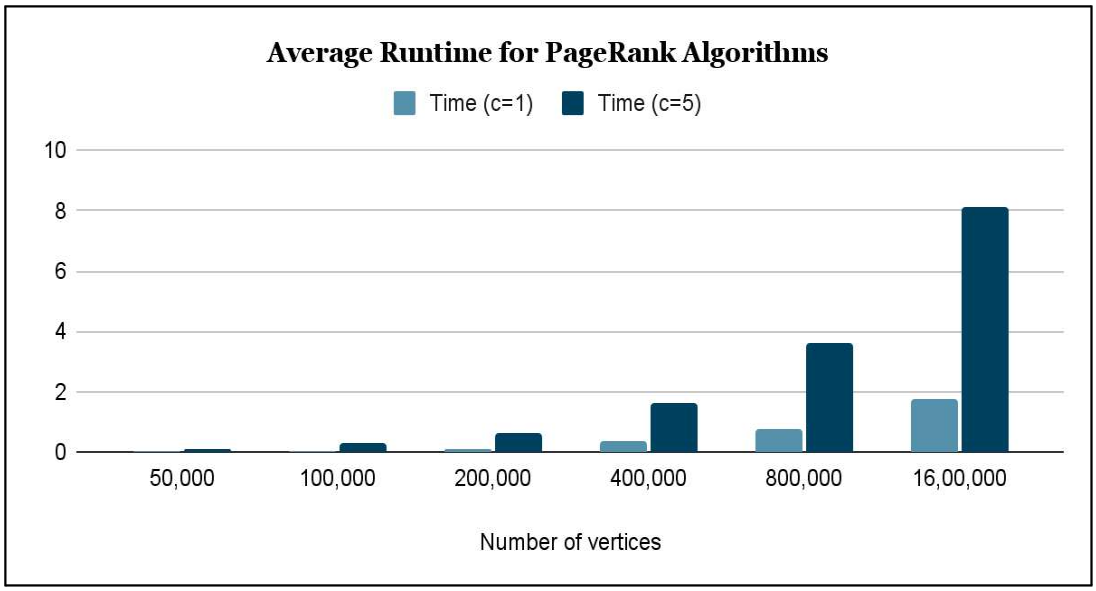
- Perform q random walks starting from the vertex.
- During each walk:
  - Increment the visit count for the current vertex.
  - Decide whether to terminate the walk based on the boredom probability (bored).
  - If the vertex is a dangling node (no outgoing links), randomly jump to another node.
  - Otherwise, randomly select one of its neighbors and continue the walk.

PseudoCode:

Algorithm 4 Monte Carlo Method (mc3) for PageRank Estimation

```
Function MonteCarloPageRank(q, m)
  Input: q (number of random walks per vertex), m (damping factor)
  Output: Approximate PageRank vector x_mc
Initialize x_mc ← zeros(num_vertices)                    ▷ Reset PageRank vector
Initialize total_visits ← 0                              ▷ Track total number of visits
for i = 0 to num_vertices − 1 do
    for dummy = 0 to q − 1 do
        w ← i                                            ▷ Start the random walk at vertex i
        while True do
            x_mc[w] ← x_mc[w] + 1                        ▷ Increment visit count for vertex w
            total_visits ← total_visits + 1              ▷ Increase total visits
            bored ← UniformRandom(0, 1)                  ▷ Check if surfer gets bored
            if bored < m then
                break                                    ▷ End walk if bored
            end if
            numNeighbors ← p[w + 1] − p[w]               ▷ Number of neighbors of vertex w
            if numNeighbors = 0 then
                w ← UniformRandom(0, num_vertices)       ▷ Jump to a random vertex
            else
                offset ← UniformRandom(0, numNeighbors)  ▷ Choose random neighbor
                w ← r[p[w] + offset]                     ▷ Move to the neighbor
            end if
        end while
    end for
end for
for i = 0 to num_vertices − 1 do
    x_mc[i] ← x_mc[i]/total_visits                       ▷ Normalize to get probabilities
end for
Output: x_mc                                             ▷ Final PageRank vector
```

Number of vertices	Time (c=1)	Time (c=5)
50,000	0.0276	0.1348
100,000	0.0584	0.2913
200,000	0.1306	0.6432
400,000	0.3359	1.6413
800,000	0.7425	3.6255
16,00,000	1.7734	8.1545



Monte Carlo Method 4: Complete Path Stopping at Dangling Nodes

- q: Number of random walks initiated from each vertex.
- m: Damping factor, representing the probability that a random walk terminates at any step (0.15).

- counter: The contribution to the PageRank value for each visit (fixed at 1.0).
- total_visits: Tracks the total number of visits across all random walks.

Monte Carlo Simulation:

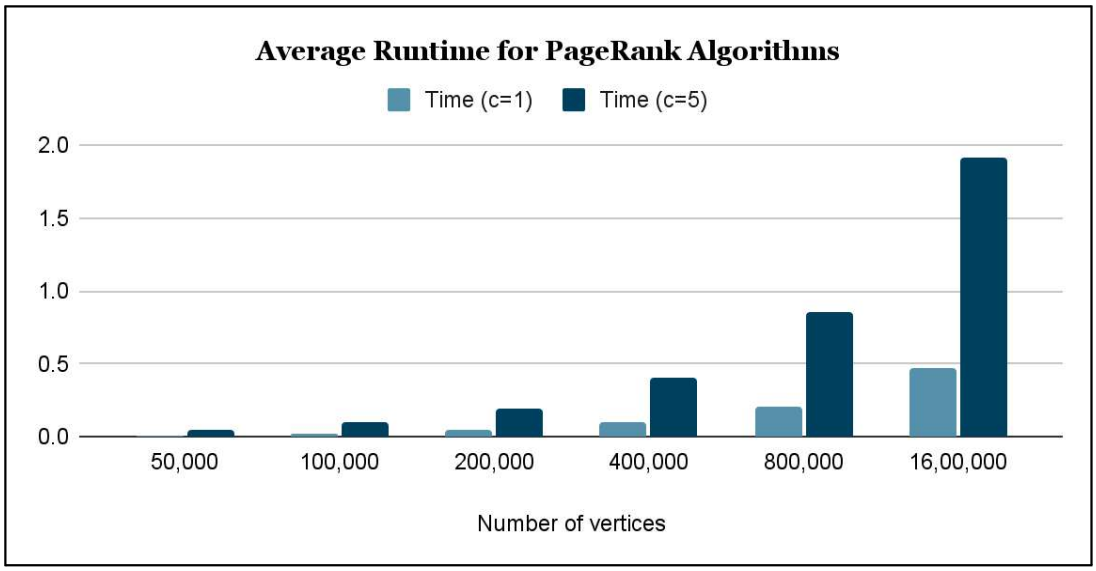
- For each vertex:
  - Perform q random walks starting from the vertex.
  - During each walk:
    - Increment the visit count for the current vertex.
    - Decide whether to terminate the walk based on the boredom probability (bored).
    - If the vertex is a dangling node (no outgoing links), terminate the walk.
    - Otherwise, randomly select one of its neighbors and continue the walk.

PseudoCode:

Algorithm 5 Monte Carlo Method (mc4) for PageRank Estimation

Function MonteCarloPageRank(q, m)
  Input: q (number of random walks per vertex), m (damping factor)
  Output: Approximate PageRank vector x_{mc}
Initialize $x_{mc} \leftarrow$ zeros($num\_vertices$) ▷ Reset PageRank vector
$counter \leftarrow 1.0$ ▷ Contribution per visit
$total\_visits \leftarrow 0$ ▷ Total count of visits
for $i = 0$ **to** $num\_vertices - 1$ **do**
  for $dummy = 0$ **to** $q - 1$ **do**
    $w \leftarrow i$ ▷ Start the random walk at vertex i
    while True **do**
      $x_{mc}[w] \leftarrow x_{mc}[w] + counter$ ▷ Increment visit count for vertex w
      $total\_visits \leftarrow total\_visits + 1$ ▷ Increase total visits
      $bored \leftarrow$ UniformRandom$(0, 1)$ ▷ Check if surfer gets bored
      if $bored < m$ **then**
        break ▷ End walk if bored
      end if
      $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of vertex w
      if $numNeighbors = 0$ **then**
        break ▷ End walk at dangling nodes
      else
        $offset \leftarrow$ UniformRandom$(0, numNeighbors)$ ▷ Choose a random neighbor
        $w \leftarrow r[p[w] + offset]$ ▷ Move to the neighbor
      end if
    end while
  end for
end for
for $i = 0$ **to** $num\_vertices - 1$ **do**
  $x_{mc}[i] \leftarrow \frac{x_{mc}[i]}{total\_visits}$ ▷ Compute the average for each vertex
end for
Output: x_{mc} ▷ Final PageRank vector

Number of vertices	Time (c=1)	Time (c=5)
50,000	0.0099	0.0471
100,000	0.0206	0.0965
200,000	0.0425	0.1939
400,000	0.0953	0.4090
800,000	0.1997	0.8536
16,00,000	0.4776	1.9153



Advantages:

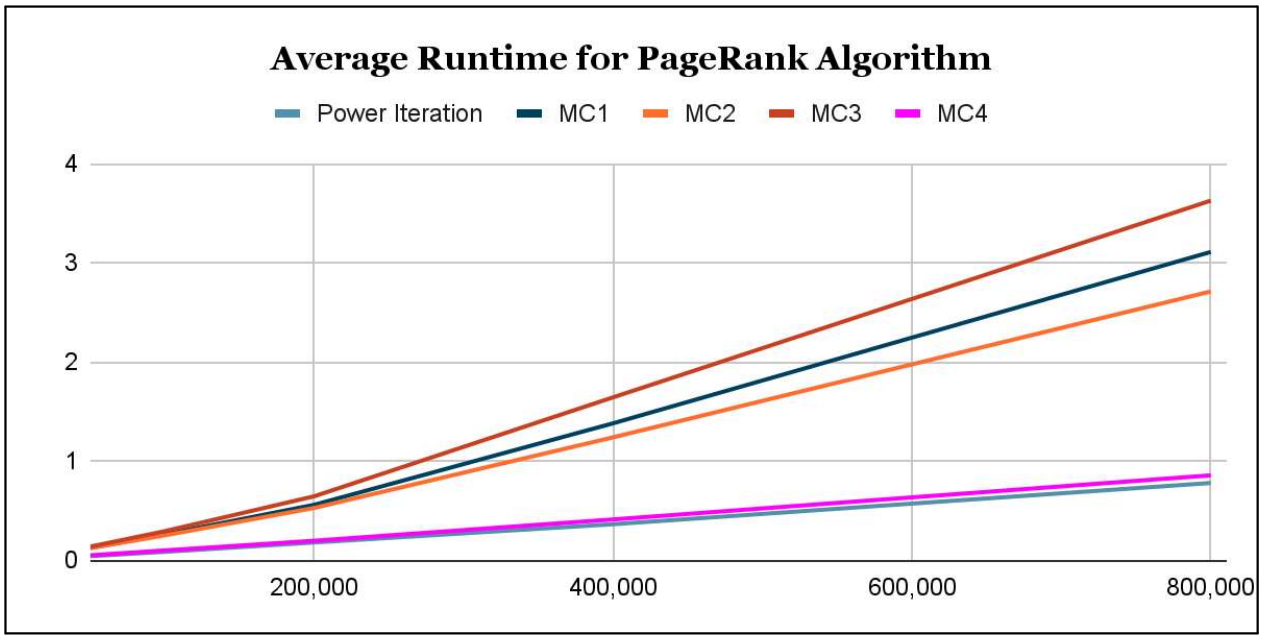
1. Monte Carlo methods are computationally efficient for very large graphs, as they avoid direct matrix operations.
2. There are several advantages of the probabilistic Monte Carlo methods over the deterministic power iteration method.
3. The algorithm is straightforward and can be parallelized easily.
4. Approximation with Fewer Resources: It provides a good approximation of PageRank without requiring full matrix computations.
5. Monte Carlo methods allow continuous update of the PageRank as the structure of the Web changes.

Challenges:

1. Accuracy vs. Efficiency Trade-off: To achieve high accuracy, a large number of random walks are needed, which can be computationally intensive.
2. Convergence Issues: Random walks may take a long time to converge for graphs with dense or highly interconnected structures.
3. Edge Cases: Handling dead ends (pages with no outbound links) and spider traps (subgraphs that are easy to get into but hard to escape) requires careful modifications, such as introducing teleportation.

It is apparent by the small relative error that the Monte Carlo methods can approximate the PageRank of the more important vertices with only one iteration per vertex in the web graph. The third and fourth Monte Carlo methods do a better job at approximating PageRank values beyond the first hundred sorted vertices, whilst the first and second Monte Carlo methods greatly grow in relative error. However, the slower runtime for the third Monte Carlo method makes the fourth Monte Carlo much more practical.

When comparing the runtimes between the four Monte Carlo methods and the Power Iteration, it turns out that the Power Iteration has faster performance overall. The fourth Monte Carlo method has a comparable runtime to the Power Iteration but the output of the Monte Carlo method is not as accurate as that of the Power Iteration.



When comparing the runtimes between the four Monte Carlo methods and the Power Iteration, it turns out that the Power Iteration has faster performance overall. The fourth Monte Carlo method has a comparable runtime to the Power Iteration but the output of the Monte Carlo method is not as accurate as that of the Power Iteration.

PageRank Simulation using Map-Reduce:

Map Reduce:
MapReduce is a programming model developed by Google for processing large datasets in a distributed and fault-tolerant manner. It splits tasks into two phases: **Map**, where input data is processed into key-value pairs, and **Reduce**, where these pairs are aggregated into the final output. MapReduce operates on a cluster of machines, leveraging distributed storage and computation to handle massive datasets efficiently. Its fault tolerance relies on re-execution of failed tasks. While powerful, it can be slower than in-memory frameworks like Spark due to its reliance on disk-based processing. It's best suited for batch processing tasks.

MRJob:
MRJob is a Python library that simplifies writing and running Hadoop MapReduce jobs. It abstracts the complexities of Hadoop streaming, enabling users to define mapper, combiner, and reducer functions easily in Python. MRJob supports running MapReduce jobs locally for testing or on Hadoop clusters for large-scale data processing. While it's disk-based and slower than Spark, it's well-suited for smaller datasets and educational purposes. MRJob is an excellent tool for beginners learning the MapReduce paradigm or prototyping Hadoop jobs.

Google Colab Notebook

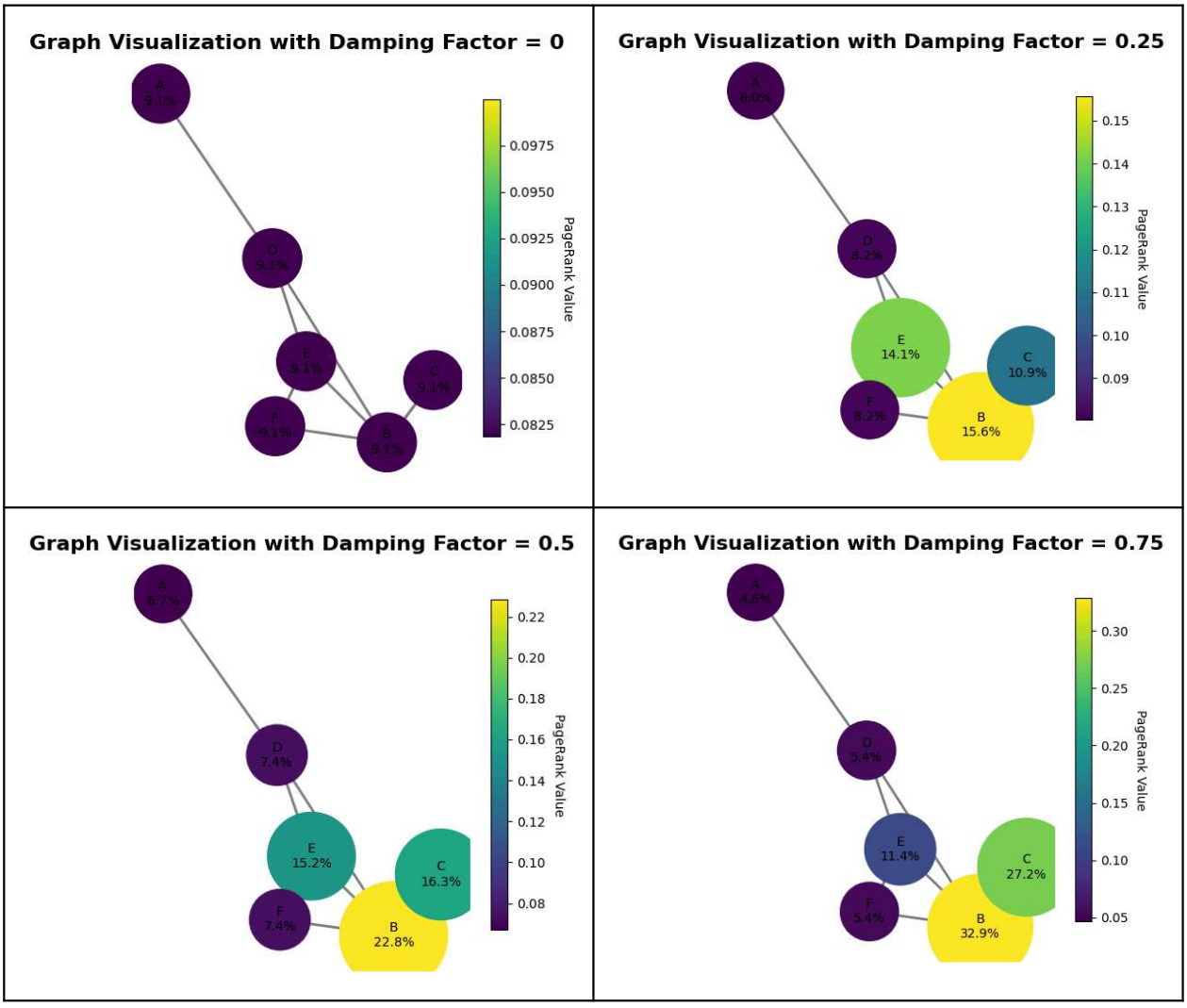
PageRank Simulation using Apache Spark:

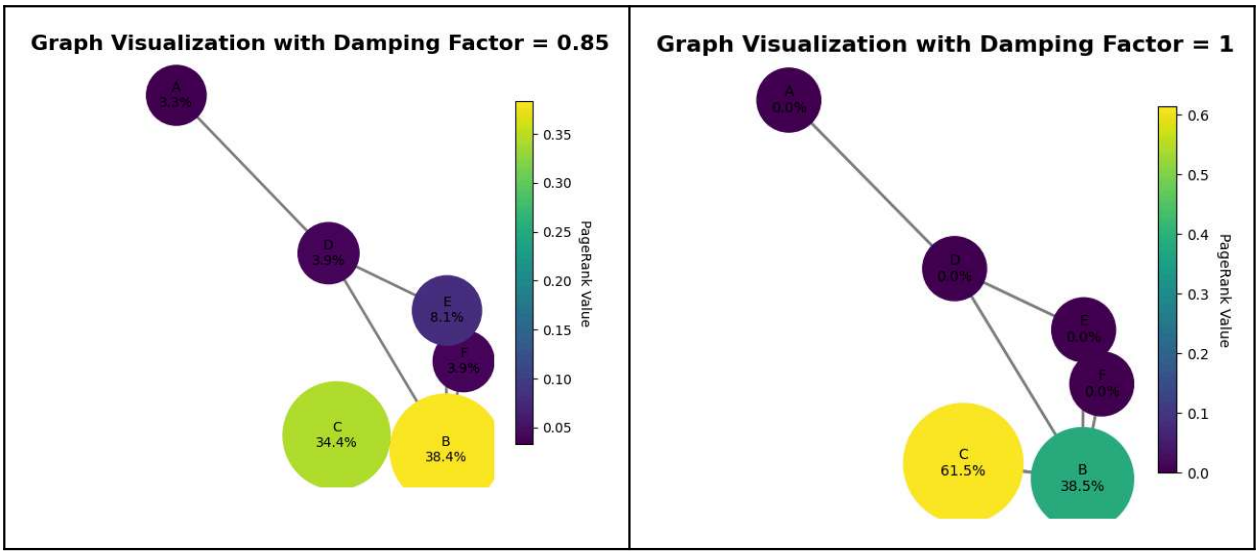
Apache Spark :
It is a powerful open-source framework for distributed data processing. It excels in in-memory computation, making it significantly faster than traditional MapReduce for iterative tasks. Spark supports diverse workloads, including batch processing, real-time streaming, machine learning, and graph processing. It provides APIs in Python, Java, Scala, and R, making it accessible to developers with varied backgrounds. Built on the concept of Resilient Distributed Datasets (RDDs), Spark ensures fault tolerance and scalability. It's ideal for large-scale analytics and high-performance applications.

Google Colab Notebook

Varying Damping Factor:

In MapReduce's PageRank, the damping factor (typically 0.85) models the probability of a random surfer continuing to click links versus jumping to a random page. It prevents overemphasis on pages with many outgoing links, ensuring stable and convergent PageRank computation by balancing random jumps and link structure, giving appropriate weight to both highly linked and less linked pages.





Observations:
As the value of the damping factor increases, the PageRank algorithm gives more weight to the link structure of the web, meaning it places greater emphasis on the probability that a random surfer will continue following links rather than jumping to a random page. A higher damping factor (closer to 1) reduces the likelihood of a random jump, making the PageRank values more sensitive to the network's link structure. This can lead to more pronounced differences in rank between highly connected pages and less connected ones, as the random jumps have less influence. However, if the damping factor becomes too high (close to 1), the algorithm might become more susceptible to rank concentration in a few well-connected pages.

Comparison Between Spark and MRJob Time Complexity:

MapReduce: 245.64 sec
Spark(w/o sorting): 16.83 sec
Spark(with sorting): 15.10 sec

Reference:

[1] Das Sarma, Atish, et al. "Fast distributed page rank computation." International Conference on Distributed Computing and Networking. Springer, Berlin, Heidelberg, 2013
[2] K. Avrachenkov et al. Monte Carlo methods in PageRank computation: When One Iteration is sufficient . In: SIAM Journal on Numerical Analysis 45.2 (2007), pp. 890 904.

Resources:

- Github Repository
- Datasets & Output Files
- Google Colab Notebook: [1] [2] [3]