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.
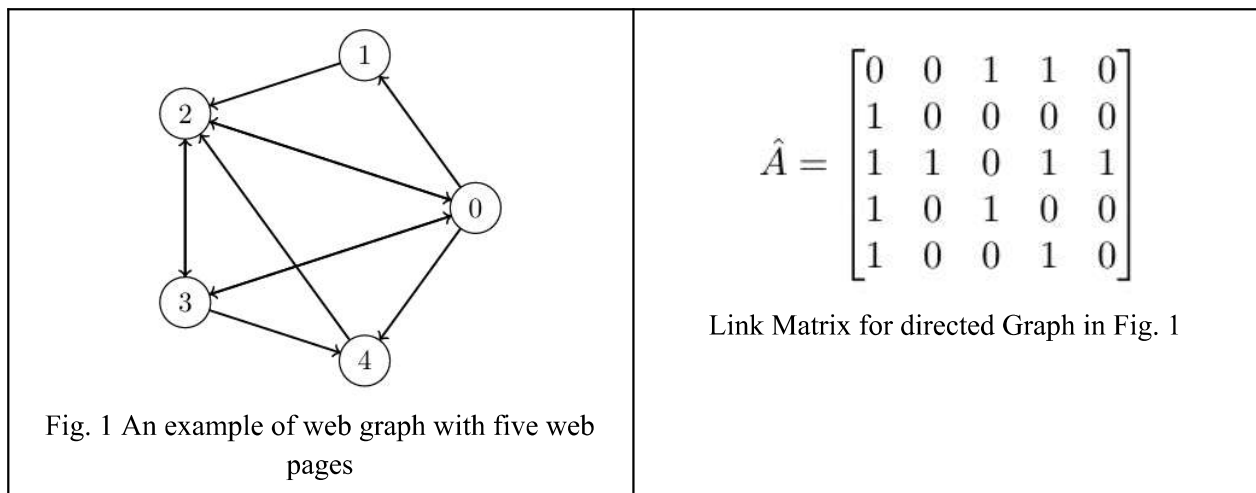
Table of Content:

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.

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

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

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

1: **Function** PowerIteration(A, x, α, ϵ, max_iters)
2: **Input:** Sparse matrix A (Google matrix), initial vector x, damping factor α, convergence threshold ϵ, maximum iterations max_iters
3: **Output:** Dominant eigenvector v, number of iterations iter_count
4: $v \leftarrow x$ ▷ Initialize vector with initial probabilities
5: $iter\_count \leftarrow 0$
6: **while** $iter_count < $ max_iters **do**
7:     $v_{new} \leftarrow \alpha \cdot A \cdot v + (1 - \alpha) \cdot x$ ▷ Compute the next vector
8:     $v_{new} \leftarrow$ Normalize(v_{new})   ▷ Normalize the vector
9:     **if** $\|v_{new} - v\| < \epsilon$ **then**   ▷ Check for convergence
10:        **break**   ▷ Exit loop if converged
11:     **end if**
12:     $v \leftarrow v_{new}$ ▷ Update the vector for the next iteration
13:     $iter\_count \leftarrow iter\_count + 1$
14: **end while**
15: **Return** $v, iter\_count$
16:
17: **Function** Normalize(v)
18: **Input:** Vector v
19: **Output:** Normalized vector v
20: $norm \leftarrow \|v\|$ ▷ Compute the norm of the vector
21: **if** $norm \neq 0$ **then**
22:     $v \leftarrow v/norm$ ▷ Normalize by dividing each element by the norm
23: **end if**
24: **Return** v
25:
26: **Main Program:**
27: Read vertex size from command-line argument
28: Create a file to log results: file = OpenFile("powertimes_".vertex_size.".txt")
29: **for** $i = 0$ to NO_OF_GRAPHS **do**
30:     Read sparse matrix from file: A $=$ LoadSparseMatrix("sparse_".i.".csv")
31:     Initialize the probability vector: $x =$ InitializeVector(vertex_size)
32:     Start timer
33:     Perform power iteration: $v, iter\_count \leftarrow$ PowerIteration($A, x, \alpha =$ $0.15, \epsilon = 1e - 8$, max_iters $= 100000$)
34:     End timer
35:     Log the results: time_elapsed, iter_count to file
36: **end for**
37: 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

Average Power Iterations Runtime
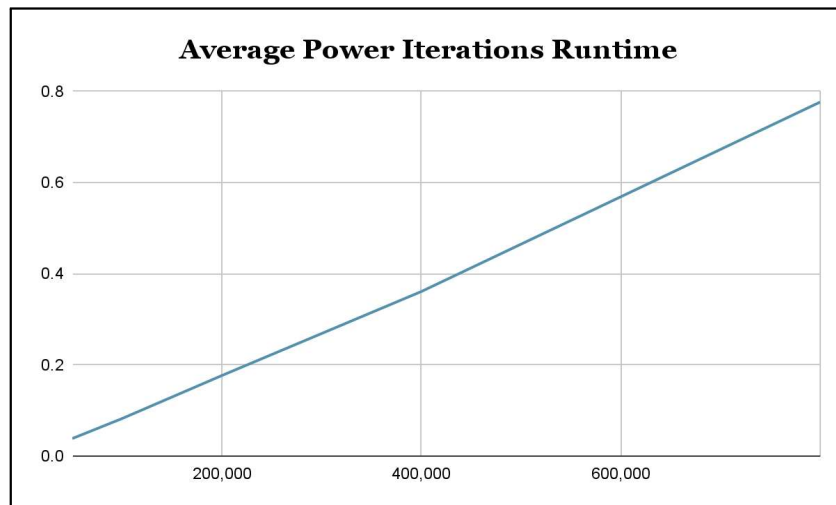
Advantages:

- **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

```
 1: Function MonteCarloPageRank(N, m)
 2:    Input: N (number of Monte Carlo simulations), m (damping factor)
 3:    Output: Approximate PageRank vector x_mc
 4: Initialize x_mc ← zeros(num_vertices)        ▷ Initialize the PageRank vector
 5: counter ← 1/N                                ▷ Counter for updating the PageRank value
 6: for i = 0 to N − 1 do
 7:    w ← UniformRandom(0, num_vertices)  ▷ Choose a random start node
 8:    while True do
 9:        bored ← UniformRandom(0, 1)  ▷ Randomly decide if the walk ends
10:        if bored < m then
11:            break                            ▷ End walk if surfer gets bored
12:        end if
13:        numNeighbors ← p[w + 1] − p[w] ▷ Number of neighbors of node w
14:        if numNeighbors = 0 then
15:            w ← UniformRandom(0, num_vertices)        ▷ Randomly jump if
    dangling node
16:        else
17:            offset ← UniformRandom(0, numNeighbors)  ▷ Pick a random
    neighbor
18:            w ← r[p[w] + offset]                     ▷ Move to a random neighbor
19:        end if
20:    end while
21:    x_mc[w] ← x_mc[w] + counter ▷ Increment PageRank estimate for node w
22: end for
23: sum ← Σ_{i=0}^{num_vertices−1} x_mc[i]             ▷ Sum all values in x_mc
24: if sum ≠ 0 then
25:    for i = 0 to num_vertices − 1 do
26:        x_mc[i] ← x_mc[i]/sum                        ▷ Normalize the PageRank vector
27:    end for
28: end if
29: Output: x_mc                                        ▷ Final PageRank vector
30:
```

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
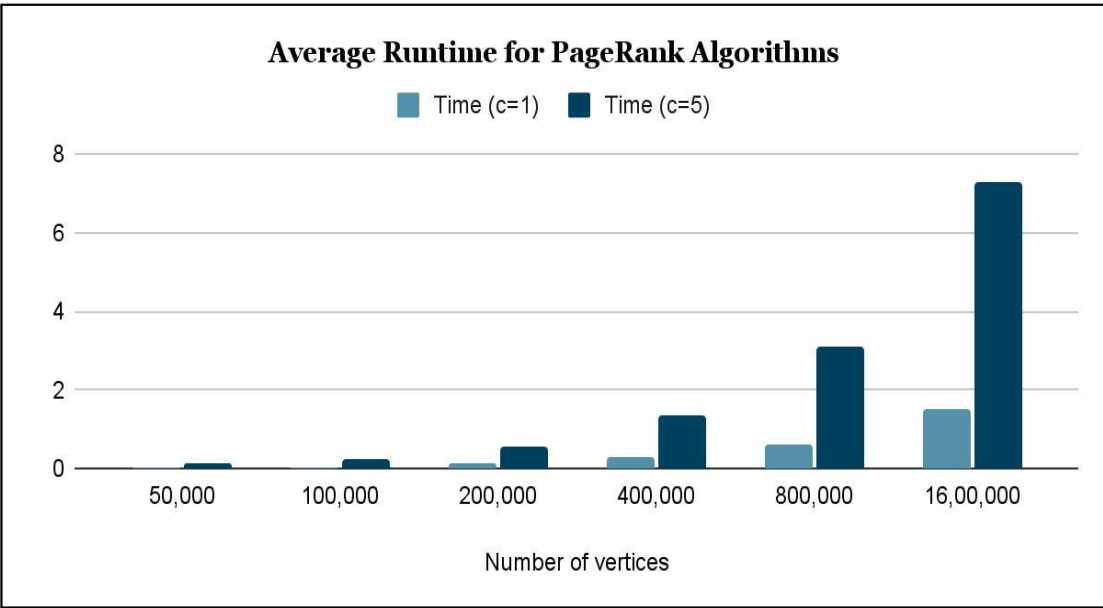
Monte Carlo Method 2: Endpoints with Cyclic Start

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.
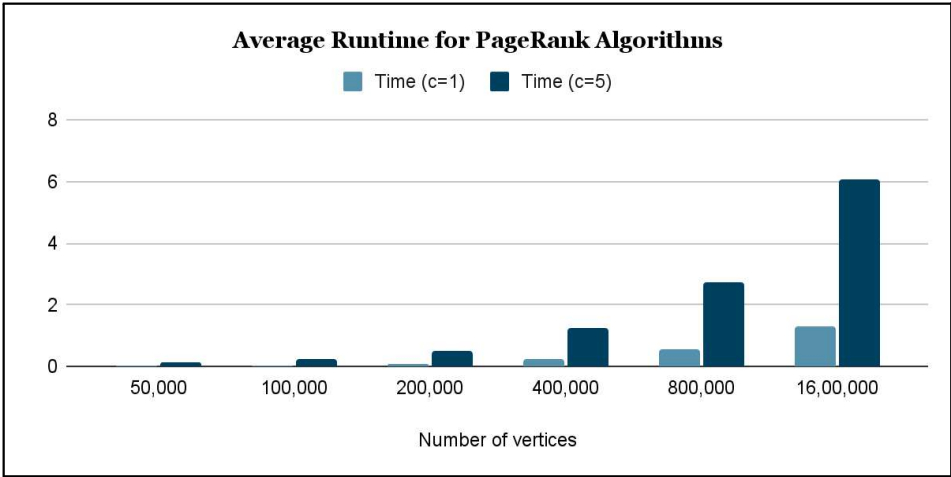
Monte Carlo Simulation:
- For each vertex:
  - Perform q random walks starting from the vertex.
  - During each walk, decide whether to terminate based on the boredom probability (bored).
  - If the vertex is a dangling node (no outgoing links), randomly jump to any node.
  - Otherwise, randomly choose 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

1: **Function** MonteCarloPageRank(q, m)
2:   **Input:** q (number of random walks per vertex), m (damping factor)
3:   **Output:** Approximate PageRank vector x_{mc}
4:   Initialize $x_{mc} \leftarrow \text{zeros}(num\_vertices)$ ▷ Reset PageRank vector
5:   $counter \leftarrow \frac{1}{q \cdot num\_vertices}$ ▷ Contribution per visit
6:   **for** $i = 0$ to $num\_vertices - 1$ **do**
7:     **for** $dummy = 0$ to $q - 1$ **do**
8:       $w \leftarrow i$ ▷ Start the random walk at vertex i
9:       **while** True **do**
10:         $bored \leftarrow \text{UniformRandom}(0, 1)$ ▷ Check if surfer gets bored
11:         **if** $bored < m$ **then**
12:           **break**   ▷ End walk if bored
13:         **end if**
14:         $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of vertex w
15:         **if** $numNeighbors = 0$ **then**
16:           $w \leftarrow \text{UniformRandom}(0, num\_vertices)$ ▷ Jump to a random vertex
17:         **else**
18:           $offset \leftarrow \text{UniformRandom}(0, numNeighbors)$ ▷ Choose random neighbor
19:           $w \leftarrow r[p[w] + offset]$ ▷ Move to the neighbor
20:         **end if**
21:       **end while**
22:       $x_{mc}[w] \leftarrow x_{mc}[w] + counter$ ▷ Increment count for ending vertex
23:     **end for**
24:   **end for**
25:   $sum \leftarrow \sum_{i=0}^{num\_vertices-1} x_{mc}[i]$ ▷ Sum all elements in x_{mc}
26:   **if** $sum \neq 0$ **then**
27:     **for** $i = 0$ to $num\_vertices - 1$ **do**
28:       $x_{mc}[i] \leftarrow \frac{x_{mc}[i]}{sum}$ ▷ Normalize to get probabilities
29:     **end for**
30:   **end if**
31:   **Output:** x_{mc} ▷ Final PageRank vector
32:

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

Average Runtime for PageRank Algorithms

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.
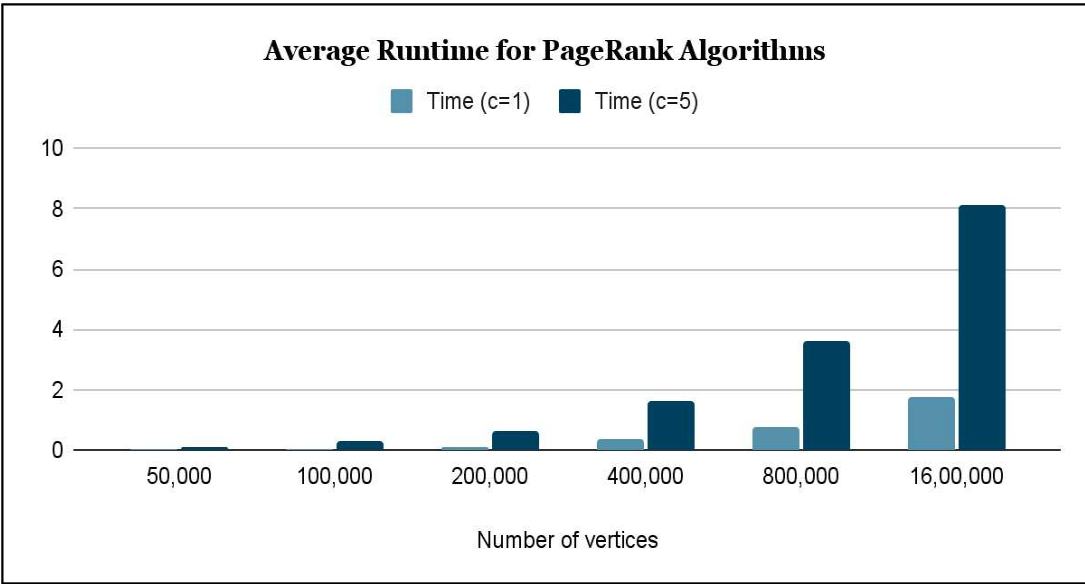
Monte Carlo Simulation:

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

1: **Function** MonteCarloPageRank(q, m)
2:     **Input:** q (number of random walks per vertex), m (damping factor)
3:     **Output:** Approximate PageRank vector x_{mc}
4: Initialize $x_{mc} \leftarrow$ zeros($num\_vertices$)   ▷ Reset PageRank vector
5: Initialize $total\_visits \leftarrow 0$ ▷ Track total number of visits
6: **for** $i = 0$ to $num\_vertices - 1$ **do**
7:     **for** $dummy = 0$ to $q - 1$ **do**
8:         $w \leftarrow i$ ▷ Start the random walk at vertex i
9:         **while** True **do**
10:           $x_{mc}[w] \leftarrow x_{mc}[w] + 1$ ▷ Increment visit count for vertex w
11:           $total\_visits \leftarrow total\_visits + 1$ ▷ Increase total visits
12:           $bored \leftarrow$ UniformRandom$(0, 1)$ ▷ Check if surfer gets bored
13:           **if** $bored < m$ **then**
14:             break   ▷ End walk if bored
15:           **end if**
16:           $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of vertex w
17:           **if** $numNeighbors = 0$ **then**
18:             $w \leftarrow$ UniformRandom$(0, num\_vertices)$ ▷ Jump to a random vertex
19:           **else**
20:             $offset \leftarrow$ UniformRandom$(0, numNeighbors)$ ▷ Choose random neighbor
21:             $w \leftarrow r[p[w] + offset]$ ▷ Move to the neighbor
22:           **end if**
23:         **end while**
24:     **end for**
25: **end for**
26: **for** $i = 0$ to $num\_vertices - 1$ **do**
27:     $x_{mc}[i] \leftarrow x_{mc}[i]/total\_visits$ ▷ Normalize to get probabilities
28: **end for**
29: **Output:** x_{mc} ▷ Final PageRank vector
30:

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

Average Runtime for PageRank Algorithms

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

1: **Function** MonteCarloPageRank(q, m)
2:   **Input:** q (number of random walks per vertex), m (damping factor)
3:   **Output:** Approximate PageRank vector x_{mc}
4: Initialize $x_{mc} \leftarrow$ zeros($num\_vertices$)   ▷ Reset PageRank vector
5: $counter \leftarrow 1.0$ ▷ Contribution per visit
6: $total\_visits \leftarrow 0$ ▷ Total count of visits
7: **for** $i = 0$ to $num\_vertices - 1$ **do**
8:   **for** $dummy = 0$ to $q - 1$ **do**
9:     $w \leftarrow i$ ▷ Start the random walk at vertex i
10:     **while** True **do**
11:       $x_{mc}[w] \leftarrow x_{mc}[w] + counter$ ▷ Increment visit count for vertex w
12:       $total\_visits \leftarrow total\_visits + 1$ ▷ Increase total visits
13:       $bored \leftarrow$ UniformRandom$(0, 1)$ ▷ Check if surfer gets bored
14:       **if** $bored < m$ **then**
15:         **break**   ▷ End walk if bored
16:       **end if**
17:       $numNeighbors \leftarrow p[w + 1] - p[w]$ ▷ Number of neighbors of vertex w
18:       **if** $numNeighbors = 0$ **then**
19:         **break**   ▷ End walk at dangling nodes
20:       **else**
21:         $offset \leftarrow$ UniformRandom$(0, numNeighbors)$ ▷ Choose a random neighbor
22:         $w \leftarrow r[p[w] + offset]$ ▷ Move to the neighbor
23:       **end if**
24:     **end while**
25:   **end for**
26: **end for**
27: **for** $i = 0$ to $num\_vertices - 1$ **do**
28:   $x_{mc}[i] \leftarrow \frac{x_{mc}[i]}{total\_visits}$ ▷ Compute the average for each vertex
29: **end for**
30: **Output:** x_{mc} ▷ Final PageRank vector
31:

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

Average Runtime for PageRank Algorithms
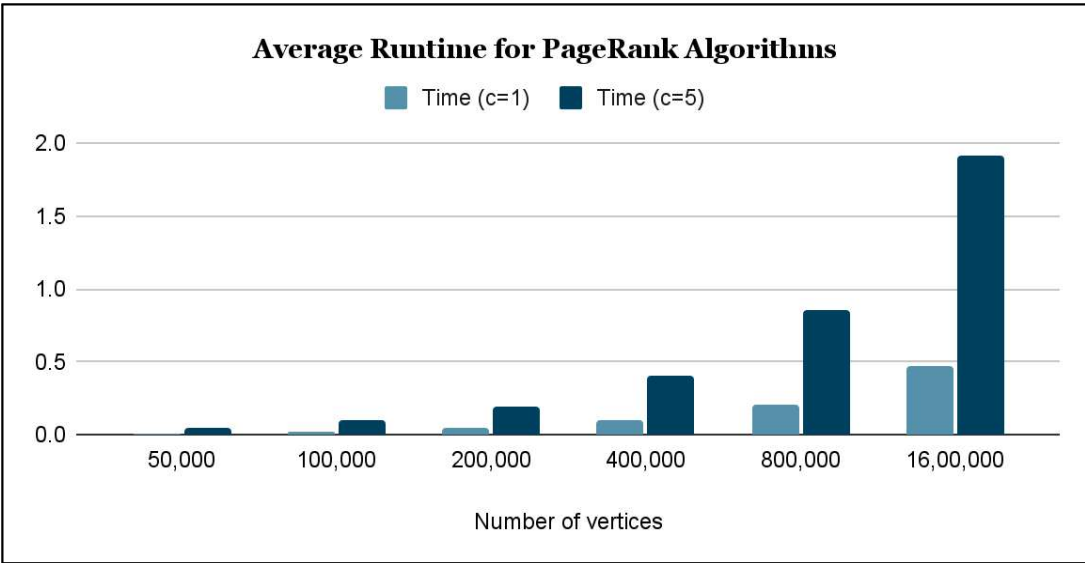
Advantages:

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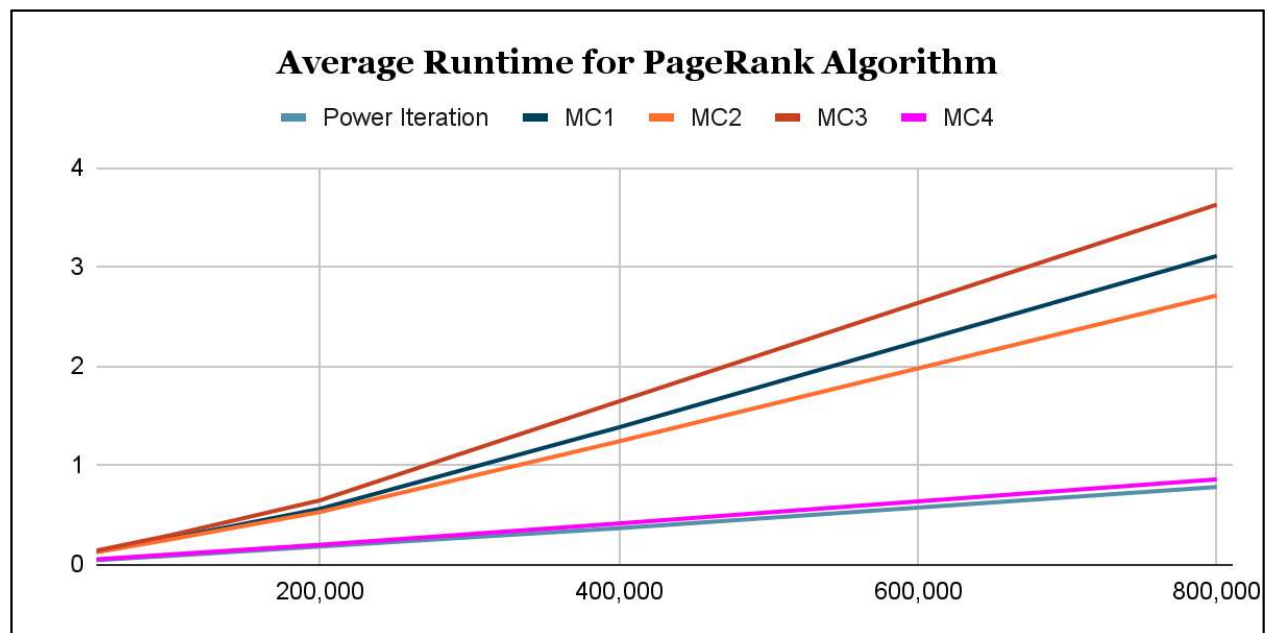
When comparing the runtimes between the four Monte Carlo methods and the Power Iteration, it turns out that the Power Iteration has faster performance overall. The fourth Monte Carlo method has a comparable runtime to the Power Iteration but the output of the Monte Carlo method is not as accurate as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

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: Whenoneiteration is su cient . In: SIAM Journal on Numerical Analysis 45.2 (2007), pp. 890 904.

Resources:
- [Github Repository](#)
- [Datasets & Output Files](#)