Analysis of the PageRank Algorithm: Computational Approaches and Performance

Angelina Cottone

Prepared for:
Professor Joseph Teran
MAT 167
18 March 2025

1 Introduction

The PageRank algorithm, developed by Larry Page and Sergey Brin at Google[1], is a quintessential process in web search ranking systems. It ranks web pages based on link structures by assigning probabilities representing the likelihood of a user landing on a given page[3]. The rank of a page is influenced by the ranks of the pages linking to it. This report aims to implement and analyze the PageRank algorithm using different computational approaches, as well as evaluate their performance on small and large-scale data.

2 Methods

2.1 Standard PageRank (pagerank1)

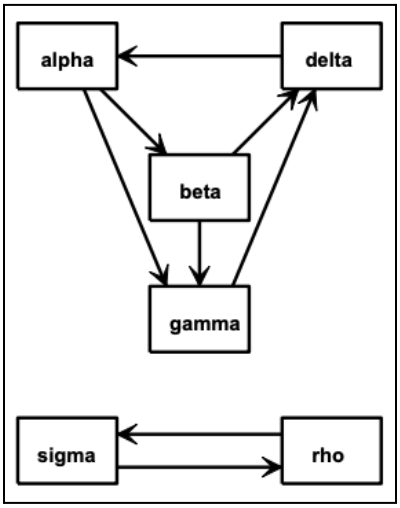
The standard PageRank method solves a linear system using 'spsolve', normalizing the solution to ensure probabilities sum to one. The key equation is $(I - pGD)x = e$, where I is the identity matrix, G is the connectivity matrix, D is the diagonal matrix with $D[i,i] = 1/c[i]$, where $c[i]$ is the sum of the column i (or 1 if column sum is 0), e is the teleportation vector where all elements equal $1/n$, and p is the damping factor.[4] The function constructs the matrices required and solves the system using sparse matrix operations. This approach is efficient for small to moderate graphs but can be computationally expensive for larger ones.

2.2 Inverse Iteration PageRank (pagerank2)

The inverse iteration method iteratively applies the stochastic transition matrix $A = pGD + \delta$ to account for teleportation, where δ is the rank-1 matrix $\delta = (1 - p)/n * ones(n, n)$, and the iteration formula is $x = Ax$.[4] The function generates a diagonal matrix D to normalize column sums. A rank vector x is initialized and a new rank vector is computed and normalized. The algorithm stops either after 100 iterations if convergence is not reached, or when the L_1 norm difference $||x_{new} - x_{old}||_1$ between iterations is less than $1e - 8$ (convergence occurs). This method is more efficient for larger-sized graphs than the direct method.

2.3 Power PageRank (pagerank3)

The power method iteratively computes the dominant eigenvector, updating as $x = (pGD)x + e(z^T x)$ [4], where z is a vector where $z[i] = 1/n$ for dangling nodes and $z[i] = 1/n - p/n$ for non-dangling nodes.[4] This function also generateds a diagonal matrix D to handle dangling nodes and starts with a uniform vector x. The z vector ensures weight is properly distributed. This approach is much more efficient for large-scale datasets, as it exclusively uses matrix-vector multiplications.



3 Results

3.1 Small Graph Analysis (Exercise 2.25)

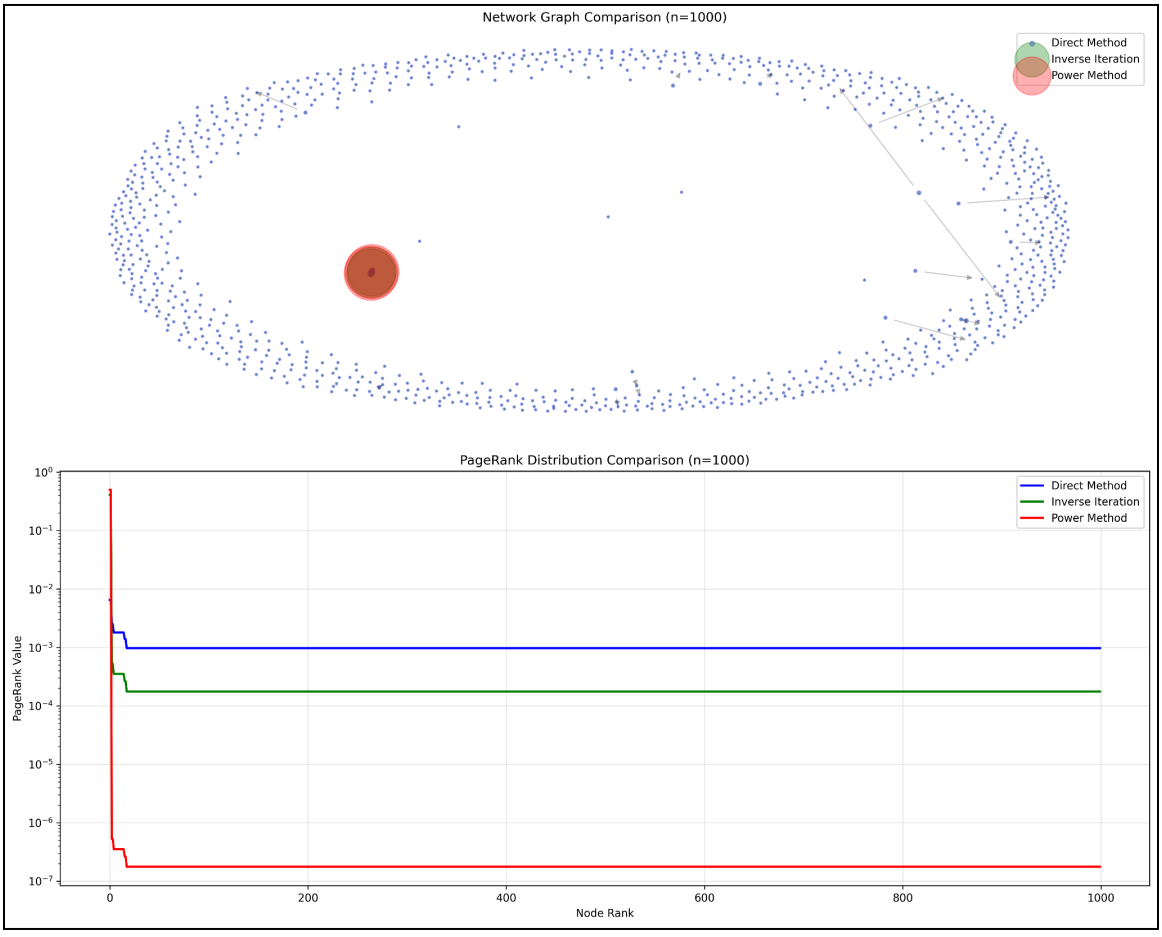
A small-scale web graph from Muler's book[1] was analyzed to understand the fundamental principles of PageRank. The structure of the graph is represented by a connectivity matrix G, where each entry G_{ij} indicates a directed link from page j to page i.

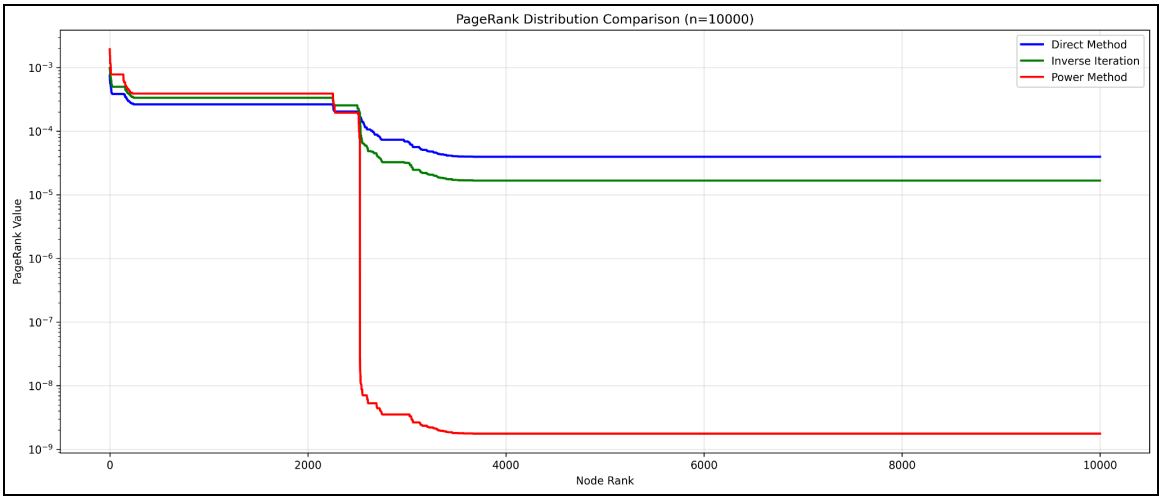
The analyzed graph consists of six nodes: alpha, beta, gamma, delta, sigma, and rho, with eight directed edges. With $p = 0.85$, nodes alpha and delta had the highest PageRank values, while node gamma had the lowest. As $p \to 1$, alpha and delta gained even higher PageRank values, while gamma further decreased. This suggests that increasing the damping factor made the network structure more influential while reducing the impact of teleportation. The power method gives slightly different results when compared to the direct and inverse methods.

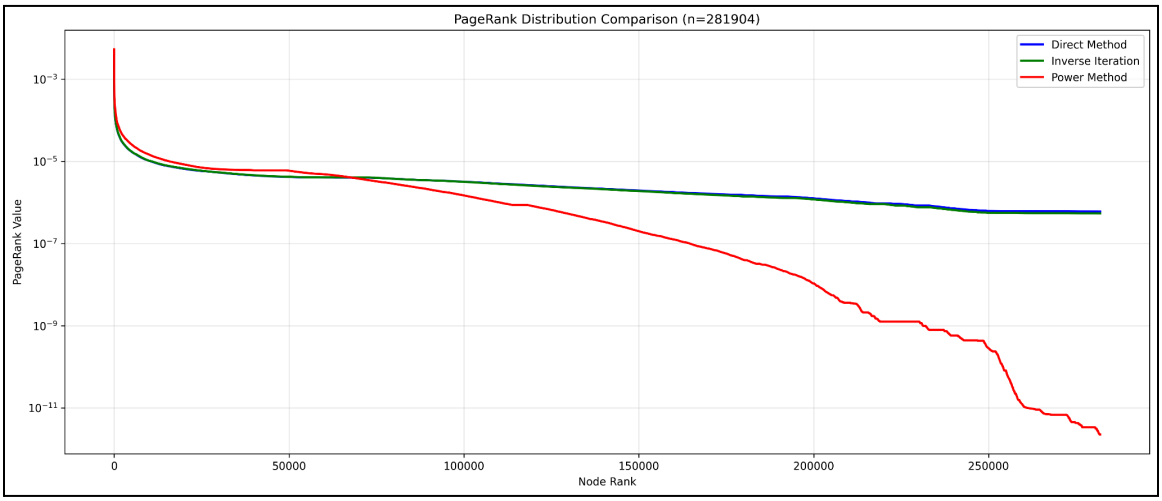
3.2 Large Dataset Analysis (web-Stanford)

The web-Stanford dataset [2] consists of 281,903 nodes and 2,312,497 edges, representing web pages from stanford.edu and the hyperlinks between them. The dataset was converted into a sparse connectivity matrix using 'scipy.sparse.csc_matrix' for efficiency. The resulting connectivity matrix G was generated so that $G[i, j] = 1$ if there is a link from node i to node j, and $G[i, j] = 0$ otherwise.

For $n = 100$, all methods produced nearly identical results. For $n = 1000$, the direct and inverse iteration methods remained consistent, but the power method significantly amplified values for highly connected nodes. For $n = 10000$ and the full dataset, similar results occurred, with the power method favoring nodes with high in-degrees, ranking them much higher than the direct and inverse iteration methods.







4 Conclusion

This analysis of the PageRank algorithm effectively identifies important nodes in networks. Higher PageRank values correspond to well-connected pages, whereas lower values indicate less influential pages. The direct and inverse iteration PageRank methods provide stable and accurate results for smaller subsets of data, but may not scale well for larger datasets. The power method proved more efficient for larger datasets but significantly amplifies the importance of in-degree nodes.

5 References

[1] C. Moler, Numerical computing with MATLAB, MathWorks, Nattick, Mass, 2004.

[2] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009.

[3] GeeksforGeeks, Page rank algorithm and Implementation, GeeksforGeeks. (2022). https://www.geeksforgeeks.org/page-rank-algorithm-implementation/ (accessed March 16, 2025).

[4] C. Moler, Experiments with MATLAB, Chapter 7: Google PageRank, MathWorks, 2007.

pagerank.py

In [ ]:
```python
import numpy as np
import time
from scipy.sparse import identity, csc_matrix
from scipy.sparse.linalg import spsolve
from visualizations import (create_visualizations)

def format_results(arr, G):
    """
    Format PageRank results.
    """
    in_degrees = np.array(G.sum(axis=0)).flatten() # Calculate sum of in-degrees (incoming links)
    out_degrees = np.array(G.sum(axis=1)).flatten() # Calculate sum of out-degrees (outgoing links)

    # List for storing results
    results = []
    for i, pr_value in enumerate(arr):
        results.append({
            'node_id': i + 1,
            'pagerank': pr_value,
            'in_degree': int(in_degrees[i]),
            'out_degree': int(out_degrees[i])
        })

    results.sort(key=lambda x: x['pagerank'], reverse=True) # Sort nodes by PageRank value in descending o
    return [f"Node {r['node_id']}: PR = {r['pagerank']:.4f}, In = {r['in_degree']}, Out = {r['out_degree']}
            for r in results]

def print_pagerank_results(results):
    """
    Print PageRank results.
    """
    for result in results:
        print(result)

def pagerank1(G, p=0.85):
    """
    Compute the PageRank of a graph.
    INPUT :
    - G : Connectivity matrix of the graph .
    - p : Damping factor ( default : 0.85) .
    OUTPUT :
    - x : PageRank vector
    """
    n = G.shape[0] # Number of nodes

    # Calculate sum of out-degrees (outgoing links)
    c = G.sum(axis=0).A[0]

    # Generate diagonal matrix D with inverse out-degrees
    D = identity(n, format='csc')
    D.setdiag(1/np.where(c != 0, c, 1)) # Use 1 for nodes with value zero to avoid division by zero

    # Uniform vector for teleportation (random jumps)
    e = np.ones(n) / n

    # Identity matrix
    I = identity(n, format='csc')

    # Solve the linear system
    x = spsolve(I - p * G @ D, e)

    return x / np.sum(x) # Normalize results to sum to 1

def pagerank2(G, p=0.85, tol=1e-8, max_iter=100):
    """
    Compute the PageRank using inverse iteration.

    INPUT:
    - G: Sparse connectivity matrix
    - p: Damping factor
    - tol: Tolerance for convergence
```

```
    - max_iter: Maximum number of iterations

    OUTPUT:
    - x: PageRank vector
    """
    n = G.shape[0] # Get number of nodes

    # Calculate sum of out-degrees (outgoing links)
    c = G.sum(axis=0).A[0]

    # Generate diagonal matrix D with inverse out-degrees
    D = identity(n, format='csc')
    D.setdiag(1/np.where(c != 0, c, 1)) # Use 1 for nodes with value zero to avoid division by zero

    # Calculate teleportation vector
    delta = (1-p)/n * np.ones(n)
    e = np.ones(n)

    # Initialize PageRank vector with uniform distribution
    x = np.ones(n) / n

    # Iterate until convergence or max iterations reached
    for _ in range(max_iter):
        x_old = x.copy()
        x = p * (G @ D @ x) + delta * np.sum(x) # Update PageRank vector
        x = x / np.sum(x) # Normalize to sum to 1

        # Check for convergence using L1 norm
        if np.linalg.norm(x - x_old, 1) < tol:
            break

    return x

def pagerank3(G, p=0.85, tol=1e-8, max_iter=100):
    """
    Compute the PageRank using power iteration.

    INPUT:
    - G: Sparse connectivity matrix
    - p: Damping factor
    - tol: Tolerance for convergence
    - max_iter: Maximum number of iterations

    OUTPUT:
    - x: PageRank vector
    """
    n = G.shape[0] # Number of nodes

    # Calculate sum of out-degrees (outgoing links)
    c = G.sum(axis=0).A[0]

    # Generate diagonal matrix D with inverse out-degrees
    D = identity(n, format='csc')
    D.setdiag(1/np.where(c != 0, c, 1)) # Use 1 for nodes with value zero to avoid division by zero

    # Initialize uniform vector
    e = np.ones(n) / n
    x = e.copy() # Initialize PageRank vector with uniform distribution

    # Compute vector z for non-dangling nodes
    z = np.ones(n) / n
    z[np.where(c != 0)[0]] -= p / n # Adjust for nodes with outgoing links

    # Iterate until convergence or max iterations reached
    for _ in range(max_iter):
        x_old = x.copy()
        x = (p * (G @ D)) @ x + e * (z @ x) # Update PageRank vector

        # Check for convergence using L1 norm
        if np.linalg.norm(x - x_old, 1) < tol:
            break

    return x / np.sum(x) # Normalize to sum to 1

def load_stanford_graph(filename):
```

```
    """
    Load Stanford web data.

    INPUT:
    - filename: Path to the data file

    OUTPUT:
    - G: Sparse connectivity matrix
    """
    with open(filename, 'r') as f:
        header_lines = sum(1 for line in f if line.startswith('#'))

    # Load edge data, skipping header lines
    edges = np.loadtxt(filename, delimiter='\t', skiprows=header_lines, dtype=np.int64)
    n = max(edges.max(axis=0)) + 1 # Calculate number of nodes

    # Create sparse adjacency matrix in CSC format
    return csc_matrix((np.ones(len(edges)), (edges[:, 0], edges[:, 1])), shape=(n, n))

def analyze_graph(G, sizes=[100, 1000, 10000, None]):
    """
    Analyze the graph using different PageRank methods.

    INPUT:
    - G: Sparse connectivity matrix
    - sizes: List of graph sizes to analyze

    OUTPUT:
    - results: Dictionary containing analysis results
    """
    # Store results
    results = {}
    computation_times = {'pr1': [], 'pr2': [], 'pr3': []}
    actual_sizes = []

    # Analyze different subsets of the graph
    for size in sizes:
        actual_size = G.shape[0] if size is None else min(size, G.shape[0])
        G_sub = G[:actual_size, :actual_size]
        actual_sizes.append(actual_size)

        size_label = "Full" if size is None else str(size)
        print(f"\nAnalyzing graph of size {size_label}...")
        print(f"Nodes: {G_sub.shape[0]}, Edges: {G_sub.nnz}")

        results[size_label] = {'stats': {'nodes': G_sub.shape[0], 'edges': G_sub.nnz}}
        pr_values_dict = {}

        # Compute PageRank using different methods
        for method_name, method_func in [
            ('pr1', pagerank1),
            ('pr2', pagerank2),
            ('pr3', pagerank3)
        ]:
            start_time = time.time()
            pr_values = method_func(G_sub)
            computation_times[method_name].append(time.time() - start_time)

            pr_values_dict[method_name] = pr_values
            results[size_label][method_name] = format_results(pr_values, G_sub)

            print(f"\nResults for {method_name}:")
            print_pagerank_results(results[size_label][method_name][:5])

        create_visualizations(G_sub, pr_values_dict, actual_size)

    return results

if __name__ == "__main__":
    labels = {
        'pr1': 'Direct Method',
        'pr2': 'Inverse Iteration Method',
        'pr3': 'Power Method'
    }
```

```
    try:
        G_stanford = load_stanford_graph('web-Stanford.txt')
        print(f"Loaded Stanford graph with {G_stanford.shape[0]} nodes and {G_stanford.nnz} edges")
        analyze_graph(G_stanford)
    except FileNotFoundError:
        print("\nStanford dataset not found.")
```

visualizations.py

In [ ]:

```
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
from scipy.sparse import csc_matrix

def create_visualizations(G, pr_values_dict, size):
    """
    Create visualizations for PageRank analysis.
    """
    fig = plt.figure(figsize=(15, 12))
    gs = plt.GridSpec(2, 1, height_ratios=[1, 1])

    method_labels = {
        'pr1': 'Direct Method',
        'pr2': 'Inverse Iteration',
        'pr3': 'Power Method'
    }

    colors = {
        'pr1': 'blue',
        'pr2': 'green',
        'pr3': 'red'
    }

    # Only create network visualization for smaller subsets
    if size <= 1000:
        ax1 = fig.add_subplot(gs[0])
        nx_graph = nx.from_scipy_sparse_array(G, create_using=nx.DiGraph)
        pos = nx.spring_layout(nx_graph) # Calculate node positions

        # Draw nodes for each PageRank method
        for method, values in pr_values_dict.items():
            node_sizes = [v * 5000 for v in values] # Adjust node sizes
            nx.draw_networkx_nodes(nx_graph, pos,
                                   node_size=node_sizes,
                                   node_color=colors[method],
                                   alpha=0.3,
                                   label=method_labels[method])

        # Draw edges with arrows
        nx.draw_networkx_edges(nx_graph, pos, alpha=0.2, arrows=True)
        ax1.set_title(f'Network Graph Comparison (n={size})')
        ax1.axis('off')
        ax1.legend()

    ax2 = fig.add_subplot(gs[1] if size <= 1000 else gs[0])

    # Plot PageRank values for each method
    for method, values in pr_values_dict.items():
        sorted_indices = np.argsort(values)[::-1]
        ax2.plot(np.arange(len(values)),
                 values[sorted_indices],
                 color=colors[method],
                 label=method_labels[method],
                 linewidth=2)

    ax2.set_xlabel('Node Rank')
    ax2.set_ylabel('PageRank Value')
    ax2.set_title(f'PageRank Distribution Comparison (n={size})')
    ax2.grid(True, alpha=0.3)
    ax2.set_yscale('log')
    ax2.legend()
    plt.tight_layout()

    if size == G.shape[0]:
```

```
        size_label = f"_{G.shape[0]}"
    else:
        size_label = str(size)

    plt.savefig(f'pagerank_analysis_comparison_{size_label}.png',
                dpi=300,
                bbox_inches='tight')
    plt.close()
```