

# BLG 223E - Data Structures

## Homework #4

Quiz date: December 30/31, 2025

PageRank: proof that eigenvectors are actually useful after all.

---

### Introduction

Modern search engines rely on algorithms that identify which pages on the web are most “important”. One of the earliest and most influential of these algorithms is PageRank, created by Larry Page and Sergey Brin in the late 1990s while they were developing Google. Their insight was that the importance of a page can be inferred from the structure of the web itself: pages that receive links from many other important pages should, in turn, be considered important. Does it sound familiar?

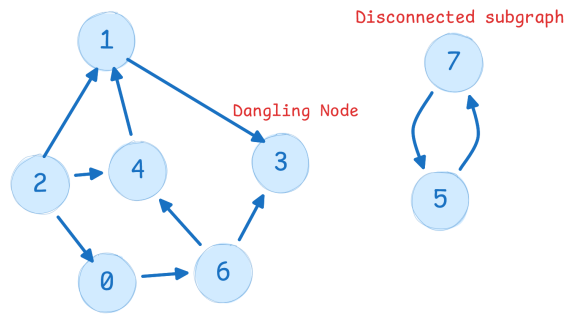
PageRank became a landmark result because it introduced a mathematically principled method for ranking billions of web pages and demonstrated that hyperlink data could be used to organize and search the web far more effectively than content-based approaches alone. Understanding PageRank provides a window into both graph algorithms and foundational ideas in network analysis.

In this homework, you will implement the simple version of the PageRank algorithm based on the random-walk-based approximation. Given the struct definitions, you will to implement the required functions. Additionally, you are asked to implement a function that finds the shortest path between given two websites using the Breadth First Search (BFS) algorithm. Finally, as a bonus task, you are asked to do the same thing using *Bidirectional* BFS algorithm.

### A Brief Mathematical Background

PageRank is a variant of **eigenvector centrality**, a measure where a node’s importance is derived from the importance of its neighbors. Hopefully, you remember this from the lecture. Formally, for a graph’s *normalized* adjacency matrix  $A$ , centrality corresponds to the eigenvector  $v$  in the equation  $Av = \lambda v$ . Unlike simple degree centrality, this recursive definition ensures that connections from high-scoring nodes contribute more significantly to a node’s score. So, the important nodes are the ones that have many important neighbors.

To apply this to the web, PageRank modifies the adjacency matrix to handle **dangling nodes** (pages with no outgoing links) and **disconnected subgraphs**, which would otherwise prevent math-



emational convergence. The algorithm introduces a damping factor  $d$  (typically 0.85) to model a “random surfer” who follows links with probability  $d$  and teleports to a random page with probability  $1 - d$ . The resulting PageRank vector is the principal eigenvector of the **Google Matrix**:

$$G = dM + (1 - d) \frac{1}{N} \mathbf{1}\mathbf{1}^T \quad ,$$

where  $M$  is the column-stochastic transition matrix, which means  $M_{ij}$  represents the probability of transitioning from website  $j$  to website  $i$ , so that the sum of the probabilities for each column is 1.

The teleportation component makes  $G$  strictly positive, guaranteeing a unique stationary distribution. In practice, this eigenvector is computed using **power iteration** ( $p_{k+1} = Gp_k$ ), which efficiently converges to the steady-state probabilities even for massive graphs.

Curious students are encouraged to look at the [original paper](#). Also, [this](#) youtube video covers the basic idea shortly.

## Implementation

You are asked to implement the PageRank algorithm using the random-walk-based approximation, instead of applying power iterations as in the original paper. So, basically, you will simulate the “random surfer model” for  $N$  steps. Notice that this is not the most efficient version of the PageRank algorithm, but it is easy to implement and it is a good way to practice graph structures.

Think of a single user surfing the web. Starting from a random website, at each step, the user either follows a link to another page or teleports to a random page. If there is no outgoing link in the current page, the user will also teleport to a random page. During surfing, you need to count the number of visits to each page. When the user finishes surfing, you need to normalize the counts to get the PageRank of each page (so that the sum of the PageRank of all pages is 1). You are asked to simulate this process for a given number of steps and calculate the PageRank of each page.

Use the following **struct** definitions and implement the graph related functions **by yourself**, such as creating a graph, adding a node, adding an edge between two specific nodes, finding a node, removing a node, removing an edge between two specific nodes, freeing the graph, and so on. You are given `web.txt` file, which represents the adjacency matrix of the graph, where each number represents the number of links from the source website to the destination website (from `row_i` to `column_j`). We also provide the expected result for the PageRank algorithm.

```

typedef struct Edge {
    struct Node *dst;
    int num_link; // there could be multiple links between two webpages
} Edge;

typedef struct Node {
    int webpage_id;
    struct Edge **adjacents;
    int num_adj;
    int capacity;
} Node;

typedef struct Graph {
    Node **nodes;
    int num_node;
    int capacity;
} Graph;

```

The expected function signature is given below. Notice that the function returns a list of the PageRank for each page. You can take `damping_factor` as 0.85, and `num_steps` as 1 million.

```
double* pagerank_random_walk(Graph *graph, double damping_factor, int num_steps)
```



**About the Memory Leaks** As a decent programmer, you need to be obsessed with the memory leaks. Please be careful about them. You can use Valgrind to check your code for memory leaks.

## Additional Task

Implement a function that returns the length of the shortest path between the two websites using the Breadth-First Search (BFS) algorithm. For the sake of simplicity, let's only use the upper-triangle part of the given matrix, and construct an undirected graph with it. The expected function signature is given below.

```

// If the destination node is not reachable, return -1.
int get_shortest_dist(Graph *graph, int src_id, int dst_id)

```



**What about DFS?** Think about the Depth-First Search (DFS) algorithm. Which one is more suitable for this problem? Can you think of a better algorithm?

Also, if you are asked to do the same task with DFS, which part of the code should you change?

## A Side Quest

This section is optional and you are not responsible for the quiz. However, If you are looking for a way to practice, you are advised to complete this section too.

*Bidirectional* BFS, is a relatively advance version of the naive BFS implementation, and as its name implies, you start the BFS from both source and target node. Somehow, its implementation might get tricky. Most of the implementations you can find online are **incorrect** (even the AI tools failed to implement this without a specific bug, but maybe the newer models can do it). Here is the [reference](#) for further details. Make sure you read the article if you want to implement this.



#### About the Usage of AI Tools

ChatGPT, Claude, and many other AI tools are powerful resources that we all have actively used in our researches and projects. However, especially at the beginning of the learning process, these tools can create an illusion of understanding. It might feel like you grasp the concepts when, in reality, you are only scratching the surface. Simply reading and momentarily understanding an AI-generated answer does not mean you truly comprehend the material.

To genuinely learn a new topic, particularly in programming courses, you need to engage with the material intentionally and put in focused effort. It is important to embrace the challenges and frustrations that come with problem-solving because this is where meaningful learning happens. AI tools can assist you along the way, but they should complement, not replace, your active participation in the learning process. So, please do not use them right away for your own sake.