Ahmed Shafiudin
Eddie Brenmark

Final Report - PageRank

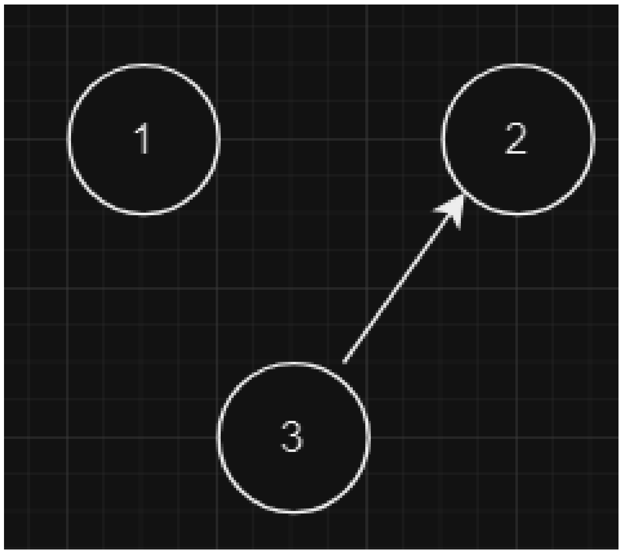
In the most general terms, the PageRank algorithm simply ranks the nodes of a directed graph. As such, one of the most obvious parameters to vary in the datasets of different graphs is the size or number of nodes in the graph. Shown below are the specific graphs we have chosen to benchmark our PageRank algorithm for.

Dataset	**Size (# of vertices)**
Graph 1 (row = 2)	10
Graph 2 (row = 8)	50
Graph 3 (row = 7)	100
Graph 4 (row = 10)	150
Graph 5 (row = 4)	200
Graph 6 (row = 1)	250

The generation of these graphs using random integers (more details in python script located within the `data` folder), ensures that our datasets capture many different instances of graph characteristics (i.e. sparse, disconnect components, etc).

To demonstrate the correctness of our implementation, it is first necessary to explain a bit about how it works. Initially, the data is read in and converted to a directed adjacency matrix. Then, it needs to be converted into a markov matrix (via normalization) where the columns of our adjacency matrix must add to 1. Finally, we need to take into account a damping factor which ensures that our random surfer will "be reset" enabling it to visit other potential connected components. Then through power iteration (matrix multiplication with a normalized tuple), we will arrive at the steady state vector which will finally hold the PageRank of all nodes. Taking a simple graph as an example: `(row1) 3vertices.csv: 0,0,2`

This can be graphically represented as follows (with adj. matrix) :



0	0	0
0	0	1
0	0	0

(*using convention where* ***from*** *col idx* ***to*** *row idx indicates 1*)

Now, converting to a markov matrix yields the following result:

1 / N = 1/3	1 / N = 1/3	0 / 1 = 0
1 / N = 1/3	1 / N = 1/3	1 / 1 = 1
1 / N = 1/3	1 / N = 1/3	0 / 1 = 0

This is calculated as follows: take the first column of the adj. matrix and normalize it. If the column sum is 0 (which it is in the first and second columns, then simply set the entries of those columns to 1 / N where N is the number of nodes (see proposal.md for a deeper explanation as to why).

Ahmed Shafiudin
Eddie Brenmark

Taking into account our damping factor (0.85), we get the following matrix and eventually the resulting vector through multiple matrix multiplications to some normalized vector in $\mathbb{R}^3$:

The specific formula is as follows:

$$\mathbf{P} = d\mathbf{M} + (1-d)\frac{1}{N}\mathbf{1}$$

“PageRank” matrix:

.85 * (⅓) + (.15)*(⅓)*(1) = 1/3	1/3	.85 * (0) + (.15)*(⅓)*(1) = 0.05
1/3	1/3	.85 * (01) + (.15)*(⅓)*(1) = 0.9
1/3	1/3	0.05

“PageRank” final vector (obtained by power iterating with above matrix with [1, 0, 0]):

0.25974026	0.48051948	0.25974026

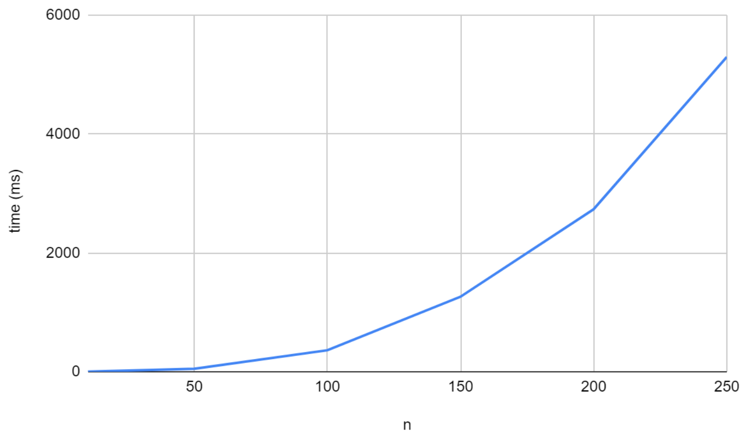
This specific test case is denoted as `"computePageRank (row 1) 3vertices"` and an additional test case breakdown can be found in the beginning of the `tests.cpp` file.

Ahmed Shafiudin
Eddie Brenmark

Big O Analysis

Now, actually putting our implementation of PageRank to the test by benchmarking it, here's what we s

n	time (ms)
10	2
50	51
100	362
150	1263
200	2736
250	5298



As one can observe, the running time of our algorithm isn't great. We can see that it models quadratic behavior and this is due to a couple reasons. One aspect which is easy to rule out is that our graph implementation of choosing adjacency matrices has no significant drawback here (other than the storage cost) since most operations used on this matrix such as checking if two vertices are adjacent are O(1). However, the initialization of this matrix proves to be costly since converting to a markov matrix requires us to iterate through each element of the matrix at least once (and hence a quadratic upper bound). Also, some of the linear algebra related functions and most notably colMatrix() proves to be yet again another expensive O(n^2) algorithm as each element in the resultant matrix needs to be modified. Optimizing these functions prove difficult as they lay the foundation for this particular implementation so trying alternative approaches to handling matrix multiplication doesn't seem feasible.

In all, however, the running time of our algorithm generally met our expectations as going in we knew that the “bottlenecks” of our program would be in the expensive mathematical functions. As a result, it would be a fair assessment to say that our algorithm matches the Big O complexity of the random surfer implementation of the PageRank algorithm based on the pseudocode on the Wikipedia page.