

CS225
Fall 2023
Extra Credit Project

Google Pagerank Algorithm

Hrishi Shah (hrishis2) and Kavya Puranam (kpura3)
Mentors Serena Trika and Blaine Hill

1. Summary of Datasets

In order to test bench the proposed algorithm, there are 5 provided CSV datasets of dimensions $(n \cdot n)$ where n is the number of columns within the database as well as the number of rows (excluding the column headers). These CSV files represent a network of sites as a graph in terms of adjacency matrices with directed relationships. The edges between nodes resemble the links between sites, and therefore they are directed as a link from site 1 to site 2 doesn't necessarily mean there is a link from site 2 to site 1. The graph is also irreflexive as most sites on the internet do not have a link to themselves. The links are randomly chosen upon creation according to the following piece of code:

```
1 import numpy as np
2 f = open("path.csv", "a")
3 n = 250
4 array = np.random.randint(low=0, high=2, size=(n, n))
5 for i in range(len(array)):
6     f.write("Site " + str(i) + ",")
7 for row in m:
8     f.write("\n");
9 for col in row:
10    f.write(col + ",")
11 print(array)
12 f.close()
```

Figure 1 - Code used to generate datasets for testing

The datasets are stored in the data folder as follows:

Name of File	Number of Sites (n)	Maximum number of edges (m , found by $n^2 - n$)
smallest.csv	5	20
second_smallest.csv	25	600
medium.csv	50	2450

second_largest.csv	100	9900
largest.csv	250	62250

Figure 2 - Summary of datasets used

Note that on average, the number of edges is half the number of maximum edges (due to the randomness pattern used to generate the matrices). Yet each dataset is still a large magnitude apart from one another. Due to the complexity of this algorithm, larger test cases were not chosen.

2. Correctness of Algorithm

To test the validity of the algorithm presented, the extensive test suite labeled as *tests.cpp* provides several requirements based on solutions generated through other means. As explored by a source in our proposal by Boston University [here](#), the popular form of calculating Pagerank right now with how large the internet has gotten is through the power iteration method (also known as the steady state method). This method takes an initial probability vector, which when given a large number of iterations will eventually represent the Pageranks of each site. The initial probability vector can therefore be anything, in our case we chose an array of all zeros with initially site 1 having a probability of 100 (as to ensure that our steady state Pageranks all add up to 100). This is the proposed method for calculating Pagerank as it works extremely well with sparse matrices, and is calculated by:

1. Taking the adjacency matrix
2. Turning it into a transition matrix (transposing and normalizing columns)
3. Conducting repeated matrix multiplication with the matrix and the probability vector
4. After enough iterations, values should steady out

This method is further explored in the next section, but for now it provides the expected Pagerank values to use in our test suite. The three highest sites are taken from the list of steady Pagerank values, and the indexes of these three highest values represent the three most important sites when it comes to ranking. The lowest site index is also stored in our solution CSVs, which are named (*dataset*)_solution.csv. Initially, the test suite was designed to check that these three indices contained the three highest values from the random surfing calculation of Pagerank. Yet due to the randomness of the algorithm, sometimes sites with very close rankings may be interchanged. Suppose that the actual rankings for the top three were that site 50 = 2.456, site 14 = 2.318, and site 9 = 2.316. Due to the randomness of the random surfer model, the top three may instead be site 50, site 9, and then site 14. It was difficult to account for this interchange, even when the number of iterations was increased to billions. So instead, we took inspiration from the test cases in mp_sketching, and used an epsilon value and made sure that our end values were within some boundary of the theoretical values. For a small dataset such as 5 sites, there was never any interchange, so we compared all five values. The process for this dataset involved:

1. Transposing the adjacency matrix
2. Normalizing the columns
3. Taking an initial probability of [100, 0, 0, 0, 0] and after repeated matrix multiplication, checking the values. These are the theoretical values for the solution used to create test cases.

```

1  # Online Python - IDE, Editor, Compiler, Interpreter
2  import numpy as np
3  import numpy.linalg as la
4  import csv
5
6  with open('data.csv', 'r') as f:
7      reader = csv.reader(f)
8      data = list(reader)
9      data_array = np.array(data, dtype=float)
10     print(data_array.shape)
11
12
13     def power_iteration(M, x):
14         # Perform power iteration and return steady state vector xstar
15         xc = x.copy()
16         for i in range(100000) :
17             #print(xc)
18             xc = M @ xc
19         return xc
20     A = data_array.T
21     print(A.shape)
22     M2 = A.copy().astype(float)
23     print(M2.shape)
24     # Convert entries in M2 below
25     for i in range(len(M2)) :
26         x = np.sum(M2[:,i])
27         if (x != 0) :
28             M2[:,i] = A[:,i] / x
29         #print(M2[:,i])
30     init = np.zeros(len(A))
31     init[0] = 100
32     print(power_iteration(M2, init))

```

Figure 3 - Code to generate steady state vector for our datasets

The values printed from this code are then used to verify that the output of our implementation provides similar results. An epsilon value is chosen, which should inversely correlate with the number of iterations (more iterations means a smaller epsilon will be fine). This value then is used similar to the delta value when calculating jaccard in mp_sketching, my making sure that the following holds: $T - \epsilon \leq R \leq T + \epsilon$ where T is the theoretical value, R is the result, and ϵ is the epsilon value. This is shown in lines 12-26 for the smallest test case in tests.cpp. It is also important to verify that the sum of all ranks should equal 100, which is the value that we chose to normalize our ranks to, which is also included within each test case by taking the sum of all ranks and ensuring that the sum is within 99.9-100.1.

3. Benchmarking

The original claim for the Big O complexity of this algorithm was that the runtime was proportional to the number of nodes in the graph + the number of edges. In the case of this implementation, this would mean that the number of iterations specified should be in a way proportional to the number of nodes + edges. Yet due to the randomness of the algorithm, it is difficult to specify a number of iterations preemptively with such a formula. Theoretically, to provide stable values it may take an infinite number of iterations. The formal proof of this algorithm with Gaussian elimination to simulate random walks provides a runtime analysis of $O(n^3)$ where n is the number of sites within the network.

Formal Big O Proof

Claim: $PR_k = \sum_j PR_j \cdot p_{ij}$ has a runtime analysis of $O(n^3)$ where PR_k is the Pagerank for a site k , and p_{ij} is the probability to travel from page i to page j . Note that n is the number of sites within the network.

Proof: Take the equation $PR_k = \sum_j PR_j \cdot p_{ij}$

This equation is self referential, so encode the rankings of all pages in the vector \mathbf{x} , and the probabilities in the stochastic matrix \mathbf{P} .

The equation can now be represented as $\mathbf{x} = \mathbf{P}\mathbf{x}$. Compute a new transition matrix by representing the damping factor. $\mathbf{P}' = d \cdot \mathbf{P} + (1 - d)/n$.

Compute a steady state for \mathbf{P}' by using $\mathbf{P}'\mathbf{x} = \mathbf{x}$ and the eigenvalue $\lambda = 1$.

To conduct Gaussian elimination for a $n \times n$ matrix, it is necessary to find n number of pivots. For each pivot it is necessary to subtract a multiple of the row from each other row which has an algorithmic efficiency of $O(n^2)$.

Therefore Gaussian elimination has an algorithmic efficiency of $O(n^3)$.

$PR_k = \sum_j PR_j \cdot p_{ij}$ has a runtime analysis of $O(n^3)$

Figure 4 - Proof of claim for eigendecomposition to provide random surfer baseline

However, this is not the method used for the implementation detailed in this report, rather just a basis for understanding steady state runtimes. With the random surfer model it is impossible to choose a formula for Big O as the randomness to the algorithm may make it difficult to choose the number of iterations necessary to return stable values. Therefore, a Big Ω is favored to provide a lower bound to the number of iterations necessary, as the pseudocode shows that the runtime is linearly dependent on the number of iterations x chosen:

- 1: declare vector to store visits
- 2: choose random site to start from
- 3: for (0, x) do
- 4: choose to either follow path or start at a new random site
- 5: increment visits
- 6: for (0, n) do
- 7: normalize the visits by the number of iterations to give percent rankings

Figure 5 - Pseudocode of Pagerank implementation

So the runtime is the max of either x or n , but should most of the time be x as x should generally be chosen to be higher than the number of sites. For example, if you have 250 sites and only choose to iterate 3 times, you would not have a proper Pagerank of all sites as most sites would be unvisited. This motivates the initial idea to choose x to be the number of sites, yet if one was to choose such a value there may still be sites that remain unvisited. Therefore, it leads us to choose a number of iterations that is dependent on both the number of nodes and the number of edges. Therefore Big Ω should be $\Omega(n + m)$ where n is the number of sites and m is the number of links. Note that at a maximum number of edges, this would mean that $\Omega(n^2)$ as there would be $n^2 - n$ number of links. When choosing this number of iterations, the runtimes can be benchmarked and plotted as follows:



n	 x	 O
5	25	0.028
25	625	0.042
50	2500	0.048
100	10000	0.053
250	62500	0.088

Figure 6 - Column 1 is number of sites, then the number of iterations , and finally the runtime

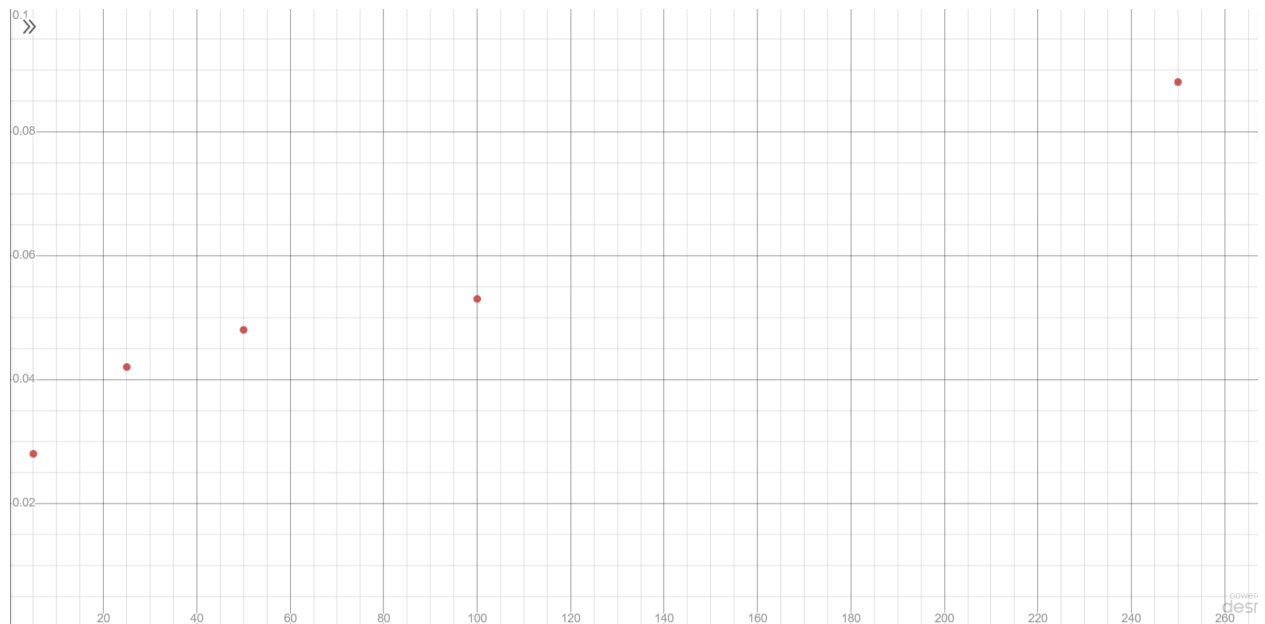


Figure 7 - Runtimes plotted, x axis is the number of sites, y axis is the runtimes in seconds

The plot alludes to potentially a logarithmic runtime, yet we know that is not possible as our code shows that the number of iterations bounds the runtime. This may be due to using actual runtimes which can differ from execution to execution.