# PageRank Algorithm

By Hrishi Shah and Kavya Puranam

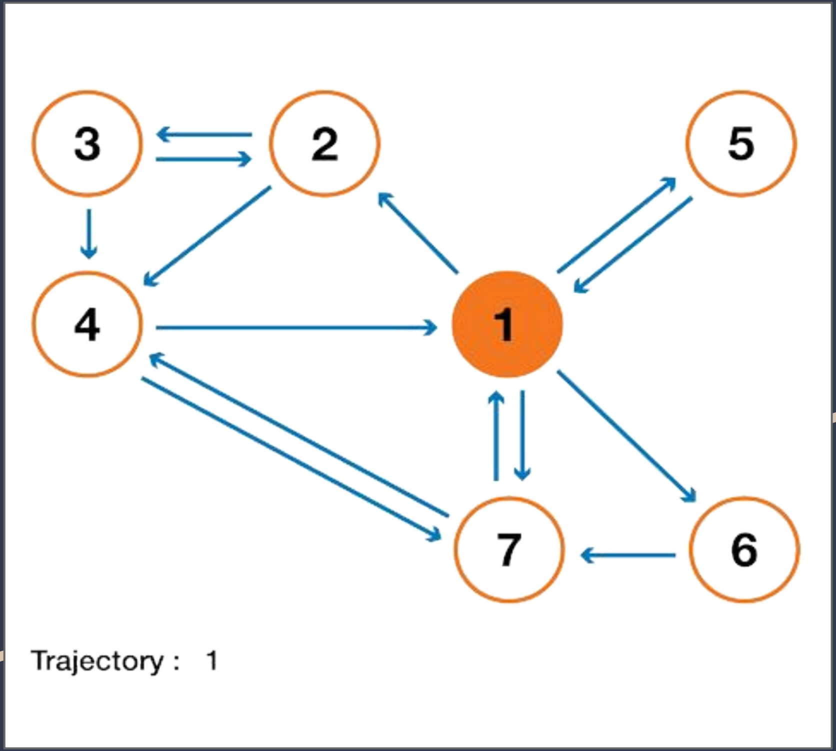
# What is the PageRank Algorithm

# PageRank

# Named after Larry Page co-founder of Google

- An algorithm used by Google Search Engine to measure the importance of a webpage
- Based on the quality and quantity of links attached to a page
  - Quality - measure of how important the pages that are connected to that page are
- Helps improve the quality of search results by bringing the most important website to the top of search engine results
- Assigns a score to each webpage based on the number of links pointing towards the webpage and importance of the web pages that link to the webpage

# Random Surfer




- Random surfing model is used on a directed graph of connected components
- This represents the network of webpages and its links
- At a certain node (webpage), the probability of the random surfer continuing to follow the links of the webpage is defined by the damping factor
- The remaining probability is that the surfer will jump to a random node
- After many iterations of the model the number of visits will result in the page's rank

# Goals

- Learn about an algorithm that is significant in our everyday lives and understand how it impacts the internet

- Explore the steps necessary to develop algorithm and test the expected behavior in order to one day create an algorithm of our own

# Development Process

# Datasets

- Datasets represent a graph of a network of sites through a directed adjacency matrix
- Removed values along the right diagonal (irreflexive graphs)
- Sites on the internet generally don't link to themselves
- Contain 5, 25, 50, 100, and 250 sites
- Compared results by checking theoretical and expected values for the three highest and one lowest ranking sites

# Adjacency Matrix

| 1 | Site 1 | Site 2 | Site 3 | Site 4 | Site 5 |
|---|---|---|---|---|---|
| 2 | 0 | 1 | 1 | 0 | 1 |
| 3 | 1 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 |
| 5 | 1 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 0 |

- Used numpy random function to generate random matrices with values 0 or 1
- Ones represent links from site row->column
- Converted the adjacency matrix to adjacency list as faster for algorithm to process.
- Took the adjacency matrix as a CSV and created an adjacency list
- Helped rid the matrix of all the zeros and compress the sparse matrices

# The Algorithm

# Output

- We used 3 functions
- **`std::vector<std::vector<int>> csvToEdgeList(std::string const & fileName)`**
  - Takes in a csv file and creates an adjacency list where we keep track of the links each site is connected to
- **`std::vector<float> getTopThreeAndLowest(std::vector<float> results)`**
  - Gets the three highest ranking pages in order and least ranking page
  - Iterates through results and finds the 3 greatest page ranks and sets the lowest rank as the last element to return
  - This was used to compared to the tests to see if the algorithm is correct, opted for epsilon values instead

# PageRank Algorithm

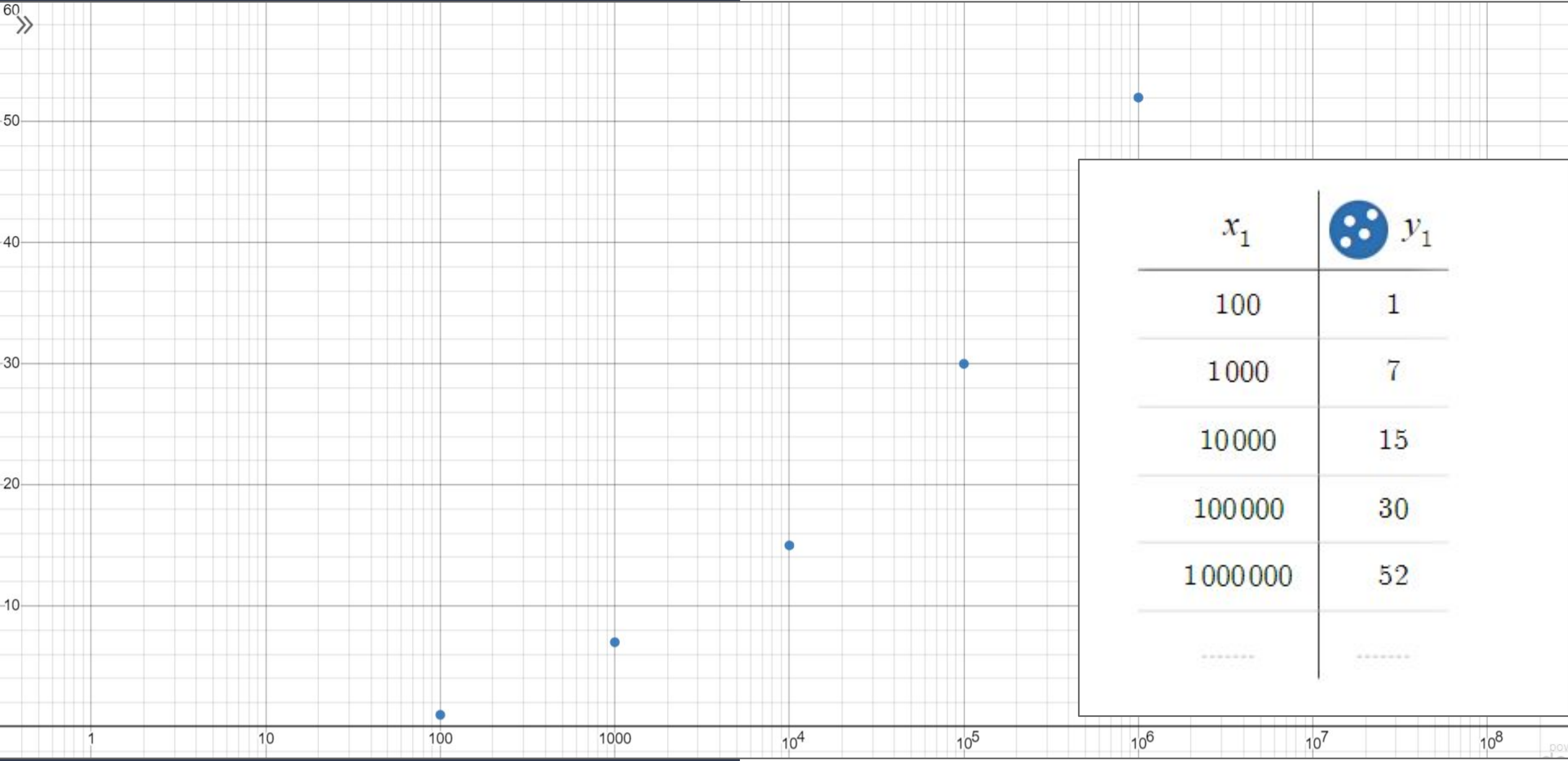
- `std::vector<float> pageRankAlgorithm(std::vector<std::vector<int>> edgeList, int n, float d)`
- Takes in a edge list, n number of pages, and d for the damping factor
- Begins on a random site in the network
- For each iteration, chooses a float value between 0-1
- Float determines whether to follow a link or “respawn” the random surfer
- Normalize the visit vectors based on number of iterations to return rankings between 0-100.

# Tests

Method:

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

- 5 tests for each of the dataset sizes:
  - Smallest (5 sites), Second Smallest (25 site), Medium (50 sites), Second Largest (100 sites), Largest (250 sites)
- Used the Power Iteration method (Steady State method) to calculate the PageRank to test our output
- Takes an initial probability vector when given a large number of iterations and will represent the Page Ranks of each site
- Initial probability vector was an array of all zeros and Site 1 is initialized to a probability of 100




| $x_1$ | $y_1$ |
|---|---|
| 100 | 1 |
| 1000 | 7 |
| 10000 | 15 |
| 100000 | 30 |
| 1000000 | 52 |

# Benchmarking results

- Original claim for Big O complexity:
  - Runtime was proportional to number of nodes in graph + number of edges
  - So number of iterations specified should be in a way proportional to the number of nodes + edges
- Due to randomness of algorithm we can’t specify a number of iterations preemptively with this formula
  - It would take an infinite number of iterations to get stable values
- Actual results showed that Big Omega* was proportional to the number of nodes and number of edges
- Differed from expectation because randomness does not necessarily account for worst case

# Big Ω:

Ω(n + m) where n is the number of sites and m is the number of links.

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

- With random surfer it is impossible to choose a formula for Big O
  - the randomness of the algorithm may make it difficult to choose the number of iterations to return stable values
- So Big Ω is favored to provide a lower bound to the number of iterations necessary

# Conclusion

# What we can improve

- Can explore the randomness of the outputs as stated before to provide better expectation for minimum iterations needed
- Can try using different methods such as power iteration and eigendecomposition
    - However eigendecomposition is more expensive at O(n^3)
- Can also extend this project by attempting the Personalized PageRank Algorithm
    - Which uses distributions biased for each individual user rather than a uniform distribution to find the rank of a page in Google search results