Marco Lipari

Programming Techniques and Applications

Robert Vincent

May 7, 2025

An Application of Markov Chains and Graph Theory to Web Page Ranking

Marco Lipari, Marianopolis College, Montreal, Quebec, Canada.

Abstract. Search engines are essential components of website navigation and web browser functionality. Notably, web-based search engines traverse the internet to generate page rankings, determining the order in which web pages appear in a search. This report focuses on applying Markov Chains and graph theory to calculate website page rankings. It explains the connection between Markov Chains and graph theory within the context of modelling web navigation. Furthermore, the report features the implementation of a web crawler that collects data for the page ranking algorithm. It also provides the software used to automate both the page ranking and web crawling processes.

Keywords: Markov Chains, graph theory, search engines, page rankings, web navigation, web crawling

Mini-Manual

Introduction. A determining factor of Google's success is the search engine's ability to feed users the most relevant results for every search. The algorithm behind this critically important system was Larry Page's PageRank algorithm, which he created in the late 90s. This project aims to use the theory behind the PageRank algorithm to rank web pages within a given domain name.

Operating Instructions. Run the file "rankalgorithm.py". Then, input the URL of the web domain to rank. Typing in 0 exits the program. Typing in 1, 2, or 3 will rank the pages on https://www.marianopolis.edu, https://quotes.toscrape.com, and https://champlainsaintlambert.ca, respectively. Files already exist for those domains, so the program skips the web scraping process. Information collected by scraping any other websites will automatically be saved. The output prints: the webpage's ranking, the webpage's URL, and then the webpage's associated importance score computed by the algorithm, on one line per page.

Requirements. The specific Python packages used in this project are noted in the "requirements.txt" file. The Python version used to develop this project is 3.13.2.

Limitations. The web crawler is a minimal implementation designed solely to support the page ranking algorithm. It is not built for large-scale deployment and has several limitations. A warning that there is no rate limiter or way to respect all robots.txt, so users should only run it if certain it respects the rules on a domain's robots.txt page.

Design Guide

Section 1

Markov Chains and Graph Theory. (Definition from Finite Mathematics class notes) A Markov Chain is a system that evolves through a series of stages; at any stage in the process, it can be in any one of a finite number of states.

The transition probability, P_{ij}, is the probability that a system in state j will transition to state i at the next stage.

Note that i) $0 \leq P_{ij} \leq 1$

ii) if $P_{ij} = 0$, then the event is impossible.

iii) if $P_{ij} = 1$, then the event is certain.

The transition matrix, P, for n states, is

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix}$$

where P is a column stochastic matrix, meaning that for column c $\in \{1, \dots, n\}$,

$$\sum_{j=1}^{n} P_{jc} = 1$$

The matrix P is analogous to an adjacency matrix, A, of a weighted directed graph with normalized edge weights, transposed. In other words, the state transition diagram of a Markov chain is an equivalent graph to the one A^T represents. For example, given

$$A^T = P = \begin{bmatrix} 0.8 & 0.1 \\ 0.2 & 0.9 \end{bmatrix}$$

The represented graph, known as the state transition diagram in the context of Markov Chains, is,

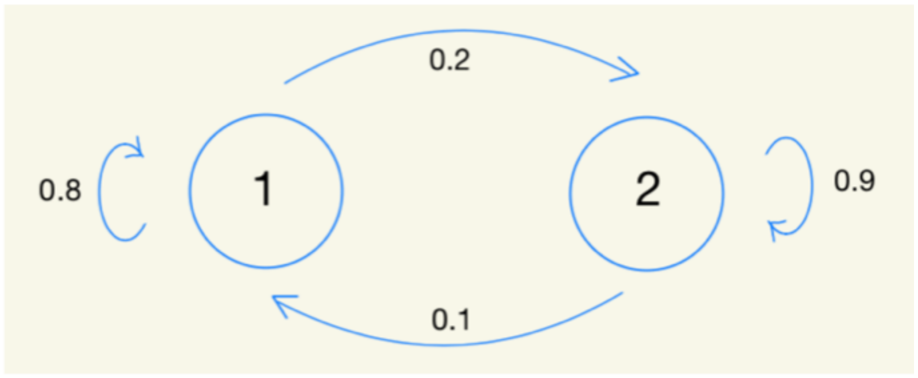

Figure from Finite Mathematics Class Notes on Markov Chains

Thus, a graph of the links between webpages can have its weights normalized to inherit the properties of Markov Chains.

A property of Markov Chains is the ability to model the long-term (or steady-state) distribution of the system. The long-term distribution vector is equivalent to the eigenvector with eigenvalue one. Thus, the long-term distribution vector, v, satisfies the equation,

$$Pv = v$$

So, v represents the distribution vector that the system always converges on after enough iterations and given any initial distribution vector.

PageRank. The PageRank algorithm models web surfing as a random walk between web pages. In this project's implementation (found in WebCrawler.py), every web page becomes a node in a graph. The graph is a directed multigraph, as it keeps track of the number of edges from a web page to another web page (i.e. if web page 1 has 3 links to web page 2 there will be 3 edges from web page 1 to web page 2). The adjacency matrix of this graph is A.

The function *DigraphToTransitionMatrix* (found in "RankAlgorithm.py") uses A to make the graph satisfy the conditions of a Markov chain. For every row of A, each component is divided by the row sum. So, the resulting row sum equals 1. But in some cases, the row sum equals 0; this row of zeros is known as a dangling vector. A dangling vector occurs when there are no outgoing links from a page; a page with no outgoing links is called a dangling node. Bringing back the random walk model of a web surfer, the web surfer would get stuck at a dangling node. But, on the web, a surfer would be able to search for other web pages using the search bar; to address this problem, the value of 1 divided by the length of the row is assigned to all components of the dangling vector. (This method of handling dangling nodes is described in equation 7 from "Toward Efficient Hub-Less Real Time Personalized PageRank" by Pirouz and Zhan linked below.) The new matrix, P, is transposed. As a result, P_{ij} represents the probability of transitioning from web page j to web page i. So, P is now a column stochastic transition matrix, modelling the process of surfing the web as a Markov Chain.

The PageRank algorithm uses the steady-state distribution vector of P to rank the importance of web pages. This works because higher-ranked pages will have many incoming links, and the pages that link to them will also have many incoming links, and so on. So, the rank of the page is determined by more than just incoming links to that page.

The method *steadystate* (found in "MatrixClass.py") computes the steady state vector using power iteration. (Power iteration is detailed in section 2.2 of "Toward Efficient Hub-Less Real Time Personalized PageRank" by Pirouz and Zhan, linked below.) The project implements power iteration in the following way: where P is the transition matrix and v_1 is a uniform distribution vector, the function first does,

$$Pv_1 = v_2$$

And then,

$$Pv_2 = v_3$$

This process of,

$$Pv_{k-1} = v_k$$

where k is the number of iterations, is repeated until,

$$Pv_k = v_k$$

with a tolerance of $1\text{x}10^{-6}$ according to the Manhattan norm between iteration k and k-1, or until k = 1000.

Notes. This implementation of PageRank does not assign a damping factor. The damping factor is a value that represents the probability of a web surfer going to a page not linked to the page they are transitioning from. This is usually necessary to prevent reducible Markov chains (when some states cannot reach other states), thus, damping ensures there is a unique steady state. But dampening is not necessary because the web crawler uses breadth-first search, therefore, all web pages must be connected to all other web pages at least through the starting web page (root node).

Section 2

DigraphStorer.py. Function *get_saved_digraph* uses built-in Python modules os, pickle, copy and urllib to save graphs so that web domains only need to be crawled once. The input is a URL, and the output is the graph object.

MatrixClass.py. Contains the class for the Matrix object used, which takes a list of lists and adds typical matrix functionality.

- size(self): Returns the matrix's dimensions as a tuple (number of rows, number of columns).
- transpose(self): Returns the transposed matrix.
- __mul__(self, other): Defines matrix multiplication.

- __pow__(self, n): Defines exponentiation of matrices.
- stochastictest(self): Verifies whether the matrix is a valid column-stochastic transition matrix (the sum of the absolute value of all the components of each column vector is ≈ 1) with a tolerance of 10^{-9}.)
- steadystate(self, tolerance = 1e-6, maxpower=1000): Computes the steady-state vector using the power iteration method, assuming the matrix is square and stochastic.
- __repr__(self): Returns a neatly formatted string representation for visualization and debugging.

RankAlgorithm.py. *DigraphToTransitionMatrix* turns digraphs into transition matrices as described in the PageRank section of the Design Guide, section 1. *Pagerank* takes the previously created digraph, computes the steady state vector, and puts the result into a list of tuples. The tuples contain each a URL, and its corresponding value from the steady state vector. This list is sorted such that the webpages are in decreasing order of importance score. The main code allows users to input their own URLs or use URLs with already saved digraphs easily.

WebCrawler.py. Contains the function *BaseUrlToDigraph*. Based on web crawler code from https://www.scrapingdog.com/blog/web-crawling-with-python/. Takes a base URL and uses breadth-first search to create a graph of its edges. The graph is a directed multigraph, meaning multiple edges are created if there are multiple equivalent outgoing links. Returns a tuple containing the adjacency matrix, and a dictionary of URLs as keys and position in the adjacency matrix as values.

WebDigraph.py. Directed multigraph class. This is a subclass of the Matrix class because it uses an adjacency matrix representation of a graph.

- __init__(self, size): Initialize an adjacency matrix of the given size.

- addEdge(self, v, w): Adds an edge from v to w. If an edge is already created, it adds another edge between the same nodes. Increases the size of the matrix if necessary.

References. Resources I used to research the topic and how they were useful.

a. Describes Markov Chains as random walks on a graph and mentions some information on page ranking. Lecture notes by Venkatesan Guruswami and Ravi Kannan, for the course Computer Science Theory for the Information Age, Spring 2012, at Carnegie Mellon University. https://www.cs.cmu.edu/~venkatg/teaching/CStheory-infoage/book-chapter-5.pdf

b. The connection between Markov Chains and Graph theory is demonstrated. "An Elementary Proof of the Markov Chain Tree Theorem" by Alex Kruckman, Amy Greenwald, and John Wicks at Wesleyan University. https://akruckman.faculty.wesleyan.edu/files/2019/07/MCTT.pdf

c. Describes PageRank from a linear algebra perspective. "The $25,000,000,000 Eigenvector: The Linear Algebra Behind Google" by Kurt Bryan, Tanya Leise. https://www.rose-hulman.edu/~bryan/googleFinalVersionFixed.pdf

d. Class notes on Markov Chains, Finite Mathematics

e. Describes Power Iteration and Dangling Nodes. Pirouz, Matin & Zhan, Justin. (2017). Toward Efficient Hub-Less Real Time Personalized PageRank. IEEE Access. PP. 1-1. 10.1109/ACCESS.2017.2773038. https://www.researchgate.net/publication/321056905_Toward_Efficient_Hub-Less_Real_Time_Personalized_PageRank#pf4 .