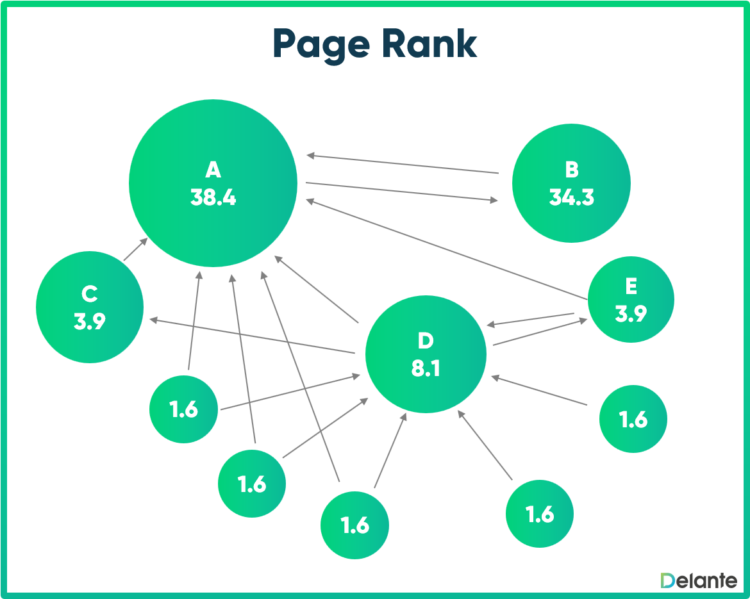
Cloud Computing Assignment 2

Hongpo Chen U2022097E

Introduction

Google's Pagerank algorithm was first developed by the founders Larry Page and Sergey Brin at Stanford University in 1998. It is a way of measuring the importance of web pages by counting the number and quality of links to each page, to estimate the importance of each website. The basis for this algorithm is the assumption that a website receiving more links from other websites is more important.[1]



A directed web graph for pagerank

Pagerank Overview and Parameters

The algorithm works by assigning a score to each web page, which represents the importance of that page. A web graph is drawn with each page acting as nodes and the interconnectedness between them as edges.

This graph is then represented in an adjacency matrix where each row represents the outgoing links and the columns represent the incoming ones. The matrix is normalized to ensure the columns add up to 1. It is *M* in the code implementation.

The probability of hopping in the graph (*d*) is utilized in this implementation of the modified pagerank, where the random surfer jumps at random. They may hop to a random page with the probability (1-*d*). *d* is set to 0.85 by default because it is the median value for the range (0.8, 0.9)

*E* is a vector for representing the likelihood of the surfer starting on each of the pages. To ensure fairness, it is usually kept uniform, meaning it is equal for all pages. Hence in the implementation, it is initialized with the function *numpy.ones().*

Experiments and Discussion

The initial 3 experiments use the 4-webpage example from the lecture notes, which is represented by the matrix  
([[0, 1/2, 0, 0],

[1/3, 0, 0, 1/2],

[1/3, 0, 1, 1/2],

[1/3, 1/2, 0, 0]])

The first run use the default parameters of 0.85 for *d* and a uniform *E* vector.

The calculated pagerank scores are 0.0825, 0.1059, 0.7058, 0.1059 for page 1, 2, 3 and 4 respectively. For the following experiments, we will observe how these values change after changing each parameters.

Experiment 2 changes *d* to 0.5, which dramatically increases the chance of a hop during random surfing. The pagerank scores are as follows: 0.1765, 0.2059, 0.4118, 0.2059. In the original web graph adjacency matrix, there exists a spider trap on node c, which causes c to have a greatly inflated score as the random surfer is likely to be stuck on it. By increasing the chance of hopping, it allows the surfer to get out of the spider trap more often and hence balances out the pagerank scores more evenly.

Experiment 3 changes *d* back to 0.85 but makes the vector *E* non-uniform. The vector is modified for simulating a situation where the webpages are for different categories which fit different users' preferences. For example, if page A is a page about sports, and the user likes sports, he is more likely to click on that than for example page B which is about video games. The vector used is 0.4, 0.3, 0.2 and 0.1 for pages A, B, C and D respectively. the pagerank scores are 0.1096, 0.1167, 0.6780, 0.0957. Because the default value is 0.25 for each of them, pages A and B have an increased pagerank score, while pages C and D have their pagerank score slightly decreased.

Experiment 4 makes changes to the adjacency matrix, by expanding it. We will see how it performs without the mapreduce. The scores are: 0.0603, 0.0942, 0.6282, 0.0774, 0.0497, 0.0657, 0.0244. The performance was not affected, which probably should not have been a surprise as it was only a 7 by 7 matrix. I took that into mind when doing the next experiment.

To simulate real world web pages which would easily have thousands of nodes in a web graph, we create a very large matrix and put random integers into it. Experiment 5 implements a thousand by thousand matrix. The results were surprisingly very uniform, with pretty much every pagerank score to be 1/1000. This is likely due to the implementation of the random matrix. It was done using *the np.random.rand* function, which may have started generating matrix elements in a similar pattern over a large number of them due to the matrix having 1 million elements(1000x1000). The performance definitely took a dip, hence a solution was needed if this algorithm was to be implemented in real world scenarios.

Implementation in MapReduce

To facilitate the large number of web pages on the internet, *MapReduce* can be used to decrease the required computation. It utilizes Mapper and Reducer functions to distribute the pagerank computation across multiple nodes. The data is divided into chunks, where the map function is executed on each input independently, creating key-value pairs. The inputs can then be shuffled, where those with the same key-value pairs are grouped together and hence reduced. By leveraging MapReduce, pagerank computations can be efficiently parallelized to handle massive web graph datasets.

An implementation would have the following steps:

1. A mapper (mapper.py)

* Reads each line and splits them into words, to create key-value pairs for each word

2. A reducer (reducer.py)

* Receives the key-value pairs from the mapper and aggregates the counts
* Outputs the final word count

3. Run the functions

* The mapper.py and reducer.py files are ran before the pagerank algorithm to parallelize computation

4. Run on hadoop

* The data could be uploaded onto HDFS for the Hadoop architecture to run the MapReduce job

A diagram of a program

Description automatically generated

MapReduce algorithm

Conclusion

The pagerank algorithm is the quite frankly the algorithm that gave rise to the internet as it is today. Life today would be hard to imagine without it. It is good to take an in-depth look into it, and find out how each of the parameters in the algorithm can be tuned for difference scenarios and use cases. As the internet is also ever-expanding with more and more users and information, we also need to consider not only the results, but also the computational efficiency of the algorithm, this is where MapReduce comes in handy.

References

1. "Facts about Google and Competition". Archived from the original on 4 November 2011. Retrieved 30 Mar 2024.

Appendix

Code for pagerank (base experiment, the other experiments are all commented out):

import numpy as np

def pagerank(M, d=0.85, num\_iterations=100):

# # experiemnt 2

# def pagerank(M, d=0.5, num\_iterations=100):

n = M.shape[0]

E = np.ones(n) / n # Initial uniform distribution vector

# experiment 3

# E = np.array([0.4, 0.3, 0.2, 0.1])

R = E.copy()

for \_ in range(num\_iterations):

R\_new = (1 - d) \* E + d \* M @ R

if np.allclose(R, R\_new):

break

R = R\_new

return R

# Example web graph (adjacency matrix)

# Rows represent outgoing links, columns represent incoming links

web\_graph = np.array([[0, 1/2, 0, 0],

[1/3, 0, 0, 1/2],

[1/3, 0, 1, 1/2],

[1/3, 1/2, 0, 0]])

# # experiment 4

# web\_graph = np.array([[0, 1/5, 0, 0, 1/4, 1/6, 1/7],

# [1/3, 0, 0, 1/2, 1/4, 1/6, 1/7],

# [1/3, 0, 1, 1/2, 1/4, 1/6, 1/7],

# [1/3, 1/5, 0, 0, 1/4, 1/6, 1/7],

# [0, 1/5, 0, 0, 0, 1/6, 1/7],

# [0, 2/5, 0, 0, 0, 1/6, 1/7],

# [0, 0, 0, 0, 0, 0, 1/7]])

# # experiment 5

# size = 1000

# shape = ((size, size))

# web\_graph = np.random.rand(\*shape)

# Normalize columns

M = web\_graph / web\_graph.sum(axis=0)

# Calculate PageRank scores

pagerank\_scores = pagerank(M)

print("PageRank scores:")

for score in pagerank\_scores:

print(f"{score:.4f},")