Numerical Analysis Project - Page Rank Implementation Using Custom Crawler

Satyajith Chilappagari (sc2100)
Mahesh Reddy Annapureddy (ma1700)
Siri Teja Chandu (sc2104)

May 2021

1 Project Description

Programming Project IV (PageRank algorithm): Page rank works by analyzing a directed graph representing the internet. Each web page is a vertex of the graph and each link is an edge. In this project you will implement a practical page rank algorithm as an iterative algorithm. PageRank works by emphasizing that the importance of a page depends only on how other pages view a page.

For example, a page that has many in-links can be considered more important than others. Similarly, if webpage A has so many visitors and if webpage A links to just one other webpage, webpage B, we can assume that many users of webpage A are likely to click on that single link B. This means that webpage B will "share" some of the importance of webpage A.

2 Implementation Link

Github repository of our implementation: https://github.com/csatyajith/weighted-page-rank

3 Introduction

Our project can be primarily divided into two parts. First is building a web index by crawling through webpages. We built a crawler using the scrapy library in python and applied transformations to it to efficiently represent over 100000 webpages.

A web crawler, spider, or search engine bot downloads and indexes content from all over the Internet. We built a web crawler that visits over 100000 pages starting from https://www.coronavirus.jhu.edu and outputs a file that contains the ids of all the webpages and the outlinks from those webpages. Then we transform this representation into a graph by letting the webpages be nodes and the hyperlinks connecting them be directed edges. While doing this, we ensure that there are no self loops and that there are no multiple edges connecting the same two webpages. Additionally, we also preserve the mappings of the webpages in a different file to refer to while recommending the top k webpages after running the weighted page rank algorithm.

Page Rank is used to rank web pages in their search engine results and it is a way of measuring the importance of website pages. It works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. Page with a link from a page which is known to be of high importance is also important.

For the purpose of this project, we study the weighted page rank algorithm. We decided to use the weighted page rank method because it assigns higher rank values to more important pages. As a result, each outlinked page gets a value proportional to the number of in-links and out-links.

4 Web Crawler

The first section is implementing a web-crawler. To do this, we used a python library called scrapy. The library allows the user to visit all the hyperlinks within a website. While the library handles visiting links, we need to write some code to reduce the size of the dataset by assigning ids to all of the samples and outputting the results into a different file. The code for this can be found below.

```
import json

import scrapy


class Spider(scrapy.Spider):
    name = "covid"
    start_urls = ["https://coronavirus.jhu.edu/"]
    COUNT_MAX = 100000

    custom_settings = {
        'CLOSESPIDER_PAGECOUNT': COUNT_MAX
    }

    def __init__(self):
        self.pc_file = open("parent_child_file.txt", "w")
        self.mapping = {}
        self.counter = 1

    def get_mapping(self, url):
        if url in self.mapping:
            return self.mapping[url]
        self.mapping[url] = self.counter
        self.counter += 1
        if self.counter == 100000:
            with open("mapping_file.json", "w") as mf:
                json.dump(self.mapping, mf)
        return self.mapping[url]

    def parse(self, response, **kwargs):
        parent = response.url
```

```
parent_mapping = self.get_mapping(parent)
yield {
    "url": response.url,
}
# now follow links to other pages
for href in response.css('li a::attr(href)').getall():
    c = parent + href[1:] if href[0] == "/" else href
    self.pc_file.write("{} {}\n".format(parent_mapping, self.get_mapping(c)))
    yield response.follow(href, callback=self.parse)
```

5 Weighted Page Rank Algorithm

In the original page rank algorithm, it is defined as the function that solves the following equation:

PageRank is introduced in the original Google paper as a function that solves the following equation:

$$PR(A) = (1-d) + d(\frac{PR(T_1)}{C(T_1)} + ... + \frac{PR(T_n)}{C(T_n)})$$

where,

- we assume that a page *A* has pages T_1 to T_n which point to it.
- *d* is a damping factor which can be set between 0 (inclusive) and 1 (exclusive). It is usually set to 0.85.
- *C(A)* is defined as the number of links going out of page *A*.

We came across the weighted page rank algorithms on the following website: Geeks for Geeks Page Rank Link. Weighted Page rank is just an extension of the Standard Page Rank Algorithm. Before we look into the formulae of weighted page rank algorithm, let us first look at some notations:

- $W^{in}_{(v,u)}$ represents the weight of the in-link
- $W^{out}_{(v,u)}$ represents the weight of the outlink
- I_p Number of inlinks of all pages cited by reference R(v)
- I_u Number of inlinks to page u
- O_p Number of outlinks of all pages cited by reference R(v)
- O_u Number of outlinks to page u

Now, we can calculate $W^{in}_{(v,u)}$ and $W^{out}_{(v,u)}$ using the following formulae:

$$W^{in}_{(v,u)} = \frac{I_u}{\sum_{p \in R(v)} I_p} \tag{1}$$

$$W^{out}_{(v,u)} = \frac{O_u}{\sum_{p \in R(v)} O_p} \tag{2}$$

Once we have these, we can calculate the page rank using the above values

$$PR_u = (1-d) + d * \sum_{v \in B(u)} PR_v * W^{in}_{(v,u)} * W^{out}_{(v,u)} \tag{3}$$

While implementing the algorithm we have to make sure to negate the effects of dead ends. Some pages are dead ends (have no links out) causing the page rank algorithm to get stuck if we land over that page. These nodes are called dead ends. To handle dead ends, we use a damping factor(β). Apart from this, we also include a teleport probability. A teleport is a condition when a user randomly moves to a different website than the current one thus resetting his traversal trace.

Initially, we compute a transition matrix M using the following method:

$$M(j,i) = \begin{cases} \frac{1}{deg(i)} & where i -> j \in E \\ 0 & otherwise \end{cases}$$

This matrix is a column-stochastic matrix. We used Power Iteration Method M where principal eigen-value of M is 1 and all other eigen-values have absolute values less than 1. Now in order to define the page rank vector we used an iterative method for finding the eigen vector corresponding to the principal eigen-value i.e 1. We define the initial page rank vector r0 as a N x 1 vector with each element equal to 1/N. We use damping factor $\beta = 0.8$. Then, our iterative formula is given by:

$$r_i = \frac{1-\beta}{N} + \beta * M * r_{i-1} \tag{4}$$

6 Implementation details

The code for matrix generation and the functions for the weighted page rank computation can be found below:

6.1 Getting the graph nodes and edges using the dataset file

```
# Code to get graph data as graph edges.
edges_of_graph = dict()

with open("page_rank/parent_child_file.txt") as file:
    line = file.readline()
    while line:
        line_list = []
        for node in line.strip().split(' '):
            line_list.append(int(node))
        if len(set(line_list)) != 1:
            if edges_of_graph.get(line_list[0]):
                edges_of_graph[line_list[0]].add(line_list[1])
```

```
            else:
                edges_of_graph[line_list[0]] = {line_list[1]}
        line = file.readline()

#print(edges_of_graph)
tot_num_of_nodes = max(sorted(edges_of_graph.keys(), reverse=True)[:1][0],
                       sorted(edges_of_graph.keys(), reverse=True)[:1][0])
print("Total number of nodes using are :- ", tot_num_of_nodes)
```

6.2 Get weighted sparse matrix from graph

```
# code to get weighted sparse matrix from graph
row_list = []
col_list = []
data_list = []

for node in edges_of_graph.keys():
    node_vals = edges_of_graph.get(node)
    node_cnt = len(node_vals)
    each_n_val = 1 / node_cnt

    values_dict = dict()
    for value in node_vals:
        if values_dict.get(value):
            values_dict[value] = values_dict.get(value) + 1
        else:
            values_dict[value] = 1
    for n_val in list(set(node_vals)):
        cnt = values_dict.get(n_val)
        data_list.append(cnt)
        row_list.append(n_val-1)
        col_list.append(node-1)

matrix = sparse.csr_matrix((data_list, (row_list, col_list)), shape=(tot_num_of_nodes,

r_list = []
c_list = []
data_list = []
r_sum = matrix.sum(axis=1).T.tolist()[0]
c_sum = matrix.sum(axis=0).tolist()[0]

for node in edges_of_graph.keys():
    node_vals = edges_of_graph.get(node)

    n_sum_in = 0
    for n_val in list(set(node_vals)):
```

```
        n_sum_in = n_sum_in + r_sum[n_val-1]

    n_sum_out = 0
    for n_val in list(set(node_vals)):
        n_sum_out = n_sum_in + c_sum[n_val-1]

    for n_val in list(set(node_vals)):
        n_in_c = r_sum[n_val-1]
        w_in = float(n_in_c/n_sum_in)
        n_out_c = c_sum[n_val-1]
        w_out =float(n_out_c/n_sum_out)
        r_list.append(n_val-1)
        c_list.append(node-1)
        weight = w_in * w_out
        data_list.append(weight)

Weighted_Sparse_Matrix = sparse.csr_matrix((data_list, (r_list, c_list)),
    shape=(tot_num_of_nodes, tot_num_of_nodes))
```

6.3 Power iteration function

```
# Function for power iteration taking matrices as input
def power_iteration(s_matrix):
    one_vec = np.ones((tot_num_of_nodes,1))
    counter = 0
    # initialise the r_0
    r_var = 1/ float(tot_num_of_nodes) * one_vec
    while True:
        temp_1 = (1-0.8)/tot_num_of_nodes
        temp_2 = 0.8 * s_matrix.dot(r_var)
        r_new_var = temp_1 * one_vec + temp_2
        L_1_Norm = np.linalg.norm(abs(r_new_var - r_var), ord=1)
        if L_1_Norm < tol_var:
            break

        r_var = r_new_var
        counter = counter + 1

    return r_var
```

7 Results

7.1 Web Crawler

The web crawler was able to traverse 100000 nodes as is evident from the run result of section 5.1. The run result can be seen in the below image:

```
Total number of nodes using are :-  108658
```

Next up we need to run our weighted page rank algorithm to get the top and bottom 10 webpages for a user's request. This numnber 10 can be customized by the user by just changing the inputs given to the function. For this, we use the following two driver functions and the below script code:

```
def getTopK(k_top,r):
    # Top k page ranks with values
    k_top_Pages_id = r.T[0].argsort()[-k_top:][::-1]
    k_top_Page_list = []
    for ind in k_top_Pages_id:
        k_top_Page_list.append((r[ind][0], ind+1))

    # Printing top k nodes with scores
    print("Top {} nodes".format(k_top))
    for pr in enumerate(k_top_Page_list):
        (ind, (pRVal, nodeId)) = pr
        print("Page rank ", ind+1, "for page id ",nodeId+1,"with value is -- ", pRVal)

    print("\n\n")

def getBottomK(k_bottom,r):
    # Bottom 5 page ranks with values
    k_bottom_Pages_id = r.T[0].argsort()[:k_bottom]
    k_bottom_Page_list = []
    for ind in k_bottom_Pages_id:
        k_bottom_Page_list.append((r[ind][0], ind+1))
    print("Bottom {} nodes".format(k_bottom))
    for pr in enumerate(k_bottom_Page_list):
        (ind, (pRVal, nodeId)) = pr
        print("Weighted page rank value", ind+1, "for page id ",nodeId+1,"is -- ", pRVa


r = power_iteration(Weighted_Sparse_Matrix)
print("weighted page rank with power iteration :- ")
getTopK(10, r)
getBottomK(10, r)
```

7.2 Top K webpages

The results of the above code look as follows.

```
Top 10 nodes
Page rank  1 for page id  108635 with value is -  3.2517930877922773e-06
Page rank  2 for page id  103571 with value is -  2.5202576326285576e-06
Page rank  3 for page id  1329 with value is -  2.322396314818146e-06
Page rank  4 for page id  75430 with value is -  2.15113725270977e-06
Page rank  5 for page id  62696 with value is -  2.0842002278284846e-06
Page rank  6 for page id  1357 with value is -  1.890483694677975e-06
Page rank  7 for page id  15617 with value is -  1.880132123692529e-06
Page rank  8 for page id  1358 with value is -  1.8795949681100727e-06
Page rank  9 for page id  25892 with value is -  1.8755128355409677e-06
Page rank  10 for page id  1361 with value is -  1.8729286630167052e-06



Bottom 10 nodes
Weighted page rank value 1 for page id  2 is -  1.840637596863553e-06
Weighted page rank value 2 for page id  72445 is -  1.840637596863553e-06
Weighted page rank value 3 for page id  72444 is -  1.840637596863553e-06
Weighted page rank value 4 for page id  72443 is -  1.840637596863553e-06
Weighted page rank value 5 for page id  72442 is -  1.840637596863553e-06
Weighted page rank value 6 for page id  72441 is -  1.840637596863553e-06
Weighted page rank value 7 for page id  72440 is -  1.840637596863553e-06
Weighted page rank value 8 for page id  72439 is -  1.840637596863553e-06
Weighted page rank value 9 for page id  72438 is -  1.840637596863553e-06
Weighted page rank value 10 for page id  72437 is -  1.840637596863553e-06
```

We performed a reverse index search to look up the webpages using the page ids. Here are five of the top 10 webpages in the above list.:

- https://www.centerforhealthsecurity.org/who-we-are/
- https://www.centerforhealthsecurity.org/news/media-coverage/
- https://www.centerforhealthsecurity.org/contact-us/
- Johns Hopkins mental health website link
- https://en.wikipedia.org/wiki/Censorship_in_the_United_States

8 Conclusion and future work

During our research, we found that apart from power iteration, we can also use other techniques such as power extrapolation to obtain the page rank. Besides this, we also observed that there was an algorithm called personalized page rank in which the results were tailored to an individual user. This can be likened to a recommendation engine and might have far reaching use cases apart from just indexing webpages.

Apart from this, a simple json searching tool can be used to extract the actual url from the page ids. We have provided the mapping between the page id and the page url on our github repository. This reverse indexing will allow for a convenient user interface for a customer to see the top ranked pages.

9 References

1. https://hippocampus-garden.com/pagerank/
2. https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/
3. https://www.geeksforgeeks.org/weighted-pagerank-algorithm/
4. http://snap.stanford.edu/data/#socnets
5. http://infolab.stanford.edu/