Page-Ranking Algorithms in Web Graphs-Hyperlink Analysis

Author: Kai Li

Professor: Dr. Narshing DEO

December 3, 2012
Contents

1 Introduction 2

2 A Brief History 2

3 The PageRank Algorithm 3
   3.1 Simplified PageRank Algorithm 3
   3.2 Special Cases 5
      3.2.1 Dangling Links 5
      3.2.2 Loops Without Outgoing Links 7
   3.3 The Complete PageRank Algorithm 8

4 Web Graph Modeling 8
   4.1 Link Structure of the Web 9
   4.2 Web Crawling 9

5 Implementation 10
   5.1 Core Data Structure 10
   5.2 Core Algorithms 10
   5.3 Function Specification 12
   5.4 Data File Format 13
   5.5 Problems and Difficulties 14
   5.6 Results 15

6 Conclusion 15

A The Complete Algorithm Code 15
1 Introduction

Back in 1990s, the occurrence of the keyword is the only important rule to judge if a document is relevant or not. The document with the highest number of occurrences of keywords receives the highest score. This is known as content-based retrieval technique. This kind of algorithms works fine in regular text retrieval, but not in the World Wide Web (WWW), which is a large-scale web with billions of web pages. In a web environment, a content-based retrieval technique is quite susceptible to simple commercial manipulation such as purposefully repeating a keyword many times within a web page. For example, one can boost the importance of a web page with respect to a query containing the keyword "university" by putting thousands of "university" within the web page, which of course would mislead the user to the incorrect result.

During the last decade, Google has become the most widely utilized search engine in the world. This success is because of the high quality results in comparison with other search engines. Among many techniques used by Google, one of the most important algorithms is the Page Ranking algorithm which use hyperlink analysis \[1\] to rate the importance of web pages and return the query results in a decreasing order of importance. Different from content-based information retrieval, hyperlink analysis considers structure (i.e. the inter-connectivity) of the web, instead of the content. In other words, the importance of a web page is determined by other web pages that it has connections to, not by itself. This type of technique has been proven to give more accurate query results in the web environment.

2 A Brief History

PageRank was developed at Stanford University by Larry Page (hence the name PageRank \[8\]) and Sergey Brin in 1996 \[9\] as part of a research project about a new kind of search engine. \[2\] Sergey Brin had the idea that information on the web could be ordered in a hierarchy by "link popularity": a page is ranked higher as there are more links to it.\[10\] It was co-authored by Rajeev Motwani and Terry Winograd. The first paper about the project, describing PageRank and the initial prototype of the Google search engine, was published in 1998;\[7\] shortly after, Page and Brin founded Google Inc., the company behind the Google search engine. While just one of many factors that determine the ranking of Google search results, PageRank continues to provide the basis for all of Google’s web search tools \[11\].

PageRank has been influenced by citation analysis, early developed by Eugene Garfield in the 1950s at the University of Pennsylvania, and by Hyper Search, developed by Massimo Marchiori at the University of Padua. In the same year PageRank was introduced (1998), Jon Kleinberg published his important work on HITS. Google’s founders cite Garfield, Marchiori, and Kleinberg in their original paper \[7\].

A small search engine called "RankDex" from IDD Information Services designed by Robin Li was, since 1996, already exploring a similar strategy for site-scoring and page ranking. \[12\] The technology in RankDex would be patented by 1999 \[13\] and used later when Li founded Baidu in China. \[14\] Li’s work would be referenced by some of Larry Page’s U.S. patents for his Google search methods[5].
3 The PageRank Algorithm

Suppose we are given a directed graph representing the structure of the web. The node in the web graph represent a web page in the World Wide Web and the directed edge indicate that there is hyperlink on the web page represented by the starting node pointing to page represented by the end node of that directed edge. Based on such a graph representation of the link structure of the web, the PageRank algorithm assign each page a credibility, which will be discussed in detail in the following sections.

3.1 Simplified PageRank Algorithm

The basic idea of PageRank is that the importance of web page depends on the pages that link to it. On the one hand, if there are many web pages linking to page \( u \), then we consider page \( u \) to be important on the web. On the other hand, if the page \( u \) has only a few pages linking to it, but those pages are authoritative ones, we also consider page \( u \) to be an important web page. In the first case, page \( u \) accumulates importance by massive collection of its incoming links. In the second case, page \( u \)'s importance is transferred from those linking to it. If we use \( r(u) \) to denote the PageRank score of page \( u \), then the above description could be expressed as the following formula:

\[
r(u_i) = \sum_{u_j \in B(u_i)} \frac{r(u_j)}{N_j}
\]

where \( B(u_i) \) is the set of web pages that points to \( u_i \) (i.e. the set of backward links) and \( N_j \) is the number of outgoing links on page \( u_j \). The matrix representation of the simplified PageRank Algorithm can be written as:

\[
R = AR
\]

where \( R = [r(u_1), r(u_2), \ldots, r(u_N)]^T \) and the terms of the matrix \( A \) are usually,

\[
a_{ij} = \begin{cases} 
\frac{1}{N_j}, & \text{if page } u_j \text{ links to page } u_i \\
0, & \text{otherwise}
\end{cases}
\]

Before we proceed to find the solution to the above equation, we would firstly like to define the term "dangling links" and "dangling nodes". Dangling links represent the links that point to the web pages that has no outgoing links and we call those pages dangling nodes. In the graph representation of the web, "dangling links" corresponds to edges pointing to those nodes with zero out degree. It’s apparent that dangling nodes will not appear on the right side of Eq. 1, thus resulting in all zero columns in the matrix \( A \). Other than those columns representing dangling nodes, the terms of each column of \( A \) sums up to 1. Matrices with all columns satisfying this property are called left stochastic matrix. In the simplified PageRank algorithm, we only consider those web graphs resulting in a left stochastic transformation matrix \( A \) (i.e. graphs that do not have nodes of zero out degree) and we leave the dangling link discussion to the next section.

With the above clarification in mind, the computation of the PageRank vector is essentially solving the linear system of Eq. 2. The form of the equation indicates that \( R \) is the eigenvector of matrix \( A \) corresponding to eigenvalue \( \lambda = 1 \) and therefore is not unique. By adding another constraint \( R^T e = 1 \), where \( e \) is the vector with all
terms to be 1, we would have a unique PageRank solution $R$ whose $L_1$ norm is 1. The remaining part of the report would make this an implicit constraint unless otherwise specified.

On condition that $A$ is left a stochastic matrix, the iterative computation of $R$ represents the evolution of a Markov Chain and the solution to $R$ is the steady state probability of the Markov Chain. The following power method is generally used to solve the problem:

$$R_{m+1} = AR_m$$

where $R_m$ is the PageRank score at the $m^{th}$ iteration and the computation can start from any non-degenerate vector.

To illustrate, consider the directed graph shown in Fig. 1 based on a tiny web. Based on the definition transformation matrix $A$, we can easily write the it as

$$A = \begin{bmatrix}
0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{3} \\
\frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} \\
\frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{2} & 0 & 0 \\
0 & \frac{1}{3} & \frac{1}{2} & 0 & 0 & 0 \\
0 & \frac{1}{3} & \frac{1}{2} & \frac{1}{3} & 0 & \frac{1}{3} \\
\frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{2} & 0 & 0
\end{bmatrix}$$

If we start from uniform distribution, the initial PageRank of each node is $\frac{1}{6}$. Let $R$ denote the initial PageRank score vector, with all entries equal to $\frac{1}{6}$. We iteratively compute the new PageRank score by multiplying the matrix $A$ to the left. Numerical computation [15] give:

$$R = \begin{bmatrix} 0.167 \\ 0.167 \\ 0.167 \\ 0.167 \\ 0.167 \\ 0.167 \end{bmatrix}, \quad PR = \begin{bmatrix} 0.264 \\ 0.111 \\ 0.139 \\ 0.125 \\ 0.222 \\ 0.139 \end{bmatrix}, \quad P^2R = \begin{bmatrix} 0.300 \\ 0.134 \\ 0.147 \\ 0.097 \\ 0.175 \\ 0.147 \end{bmatrix}, \quad \cdots, \quad P^{13}R = \begin{bmatrix} 0.265 \\ 0.138 \\ 0.150 \\ 0.110 \\ 0.187 \\ 0.150 \end{bmatrix}$$

We observe that the sequence of iterations $R, PR, P^2R, \cdots, P^nR$ tends to con-
verge to the value $R^* = \begin{pmatrix} 0.265 \\ 0.138 \\ 0.150 \\ 0.110 \\ 0.187 \\ 0.150 \end{pmatrix}$, which is the solution to the PageRank of all web pages.

The convergence of the above method is guaranteed as long as the matrix $A$ is a left stochastic matrix.

### 3.2 Special Cases

Until now, we only give a simplified version of the PageRank algorithm and have not considered the effect of some special cases. We would see that in the presence of dangling links and loops with no outgoing links, the simplified algorithm would experience some problems, and certain modification should be made to make the algorithm applicable to all kinds of situations.

#### 3.2.1 Dangling Links

As explained in previous sections, dangling links are those links pointing to dangling web nodes. In reality, most dangling links are those pages that have not been downloaded yet. Dangling links are unavoidable because the web graph is huge and dynamic, and it is impossible to download all of them for analysis. Those dangling links do have an effect on the simplified PageRank algorithm we discussed before. To illustrate the problem, let us consider the simple web graph shown in Fig. 2. The transformation matrix is

$$A = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{bmatrix}$$

Figure 2: A tiny web graph with a dangling node

Note that the column representing node $c$ are all zeros because $c$ does not have outgoing links. If we apply the power method computation to a uniform initialization
\[ R = \begin{pmatrix} 0.333 \\ 0.333 \\ 0.333 \end{pmatrix}, \text{ we would get the sequence of } R \text{ vectors as follows:} \]

\[
\begin{pmatrix} 0.167 \\ 0.167 \\ 0.333 \end{pmatrix}, \begin{pmatrix} 0.0835 \\ 0.0835 \\ 0.167 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
\]

As we can see from this example, the dangling nodes get PageRank scores from backward links at every step, but they never contribute the score in the computation, resulting a steady loss of total PageRank score at every iteration. Therefore, the simplified PageRank algorithm could not be used for the computation. There are different ways of dealing with dangling nodes. One of the most widely used method is to evenly distribute the score of the dangling nodes to all of the nodes, assuming they have a link to all of the nodes in the web graph. The modified PageRank algorithm is:

\[
r(u_i) = \sum_{u_j \in B(u_i)} \frac{r(u_j)}{N_j} + \sum_{u_j \in D} \frac{r(u_j)}{N} \tag{4}
\]

where \( D \) is the set of all dangling nodes and \( N \) is the total number of nodes in the web graph considered. Some literature also write Eq. 4 as:

\[
r(u_i) = c \sum_{u_j \in B(u_i)} \frac{r(u_j)}{N_j} \tag{5}
\]

where \( c \geq 1 \) is a normalization constant to make the sum of all PageRank scores equal to 1. This expression is equivalent to Eq. 4 and it’s easy to verify that

\[
c = \frac{1}{1 - \sum_{u_j \in D} \frac{r(u_j)}{N}}
\]

Correspondingly, the terms of the new transformation matrix \( A \) are

\[
a_{ij} = \begin{cases} 
\frac{1}{N_j}, & \text{if page } u_j \text{ links to } page u_i \\
\frac{1}{N}, & \text{if page } j \text{ is dangling} \\
0, & \text{otherwise}
\end{cases}
\]

Now we apply the changes to example shown in Fig. 2, we can write the new matrix as

\[
A = \begin{bmatrix} 
0 & \frac{1}{2} & \frac{1}{3} \\
\frac{1}{2} & 0 & \frac{2}{3} \\
\frac{1}{2} & \frac{1}{2} & 0
\end{bmatrix}
\]

Now \( A \) remains a left stochastic matrix and by applying the power method we discussed before, the solution to \( R \) is

\[
R^* = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}
\]
3.2.2 Loops Without Outgoing Links

Loops are formed by a group of web nodes with no outgoing links (see Fig. 3). Once we get into the loop, we will never get out by following the links on the web page that belongs within the loop. Again consider the example shown in Fig. 4. The transformation matrix $A$ is written as

$$A = \begin{bmatrix}
0 & \frac{1}{2} & 0 \\
\frac{1}{2} & 0 & 0 \\
\frac{1}{2} & \frac{1}{2} & 1
\end{bmatrix}$$

![Figure 3: A loop in the web](image)

![Figure 4: A tiny web graph with a loop](image)

A good thing about the loop is that it does not break the left stochastic property of $A$, so we use the previous power method to iteratively evaluate the value of $R$. Starting from $R = \begin{pmatrix} 0.167 \\ 0.167 \\ 0.167 \end{pmatrix}$ and evaluate $A \cdot R$ in every iteration give us the following sequence of vector

$$
\begin{pmatrix}
0.333 \\
0.333 \\
0.333
\end{pmatrix},
\begin{pmatrix}
0.0835 \\
0.0835 \\
0.833
\end{pmatrix}, \ldots,
\begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix}
$$

As we can see, the loop will accumulate rank but never distribute any rank (since there are no outedges). The loop forms a sort of trap which is usually called a rank...
sink. To overcome the problem of rank sinks, a rank source $E(u)$ is introduced. It's weighted combination with the original PageRank is given as the final PageRank

$$r(u_i) = d\left( \sum_{u_j \in B(u_i)} \frac{r(u_j)}{N_j} + \sum_{u_j \in D} \frac{r(u_j)}{N_j} \right) + (1 - d)E(u_i)$$  \hspace{1cm} (6)$$

where $d \in [0, 1]$ is the damping factor and can be seen as the weighting factor between the PageRank contributed by hyperlinks and that by rank source. By requiring that the sum over all web pages $\sum_u E(u) = 1$, $E(u)$ can be seen as the probability distribution of a random jump to web page $u$. The introduction of the rank source term can resolve the rank sink issue because it provides a probability of jumping out of current loop, which would otherwise be impossible if the surfer keep following the hyperlinks on the web page he visits.

Different $E(u)$ correspond to different surfer model. The most simple one is to give every web page equal weights, resulting $E(u) = \frac{1}{N}$ for all web page $u$, which is called random surfer model. Random surfer model models the behavior that a surfer when gets bored by successively clicking on links of currently visiting web pages has the same probability of jumping to any page on the web. To give another example, by setting the weight of a particular web page to 1 and all the rest to be 0 models the behavior that a surfer always jumps to a particular web page when he gets bored on current page. This particular page can be seen as what we normally use as home page.

### 3.3 The Complete PageRank Algorithm

After we considered dangling links and loops, the complete PageRank algorithm is given in Eq. 6. The corresponding matrix form is

$$R = AR + E$$  \hspace{1cm} (7)$$

where $E = [E(u_1), E(u_2), \cdots, E(u_N)]^T$, and $A$ and $R$ are as explained in previous sections. Basic linear algebraic solution to Eq. 7 will give us

$$R = (I - A)^{-1}E$$

However, the above solution is generally not used for computation because when it comes to large graphs, getting matrix inverse is rather computationally expensive and is not easy to be implemented. In comparison, the power method is an iterative computation process and is very easy to be written as computer programs, and is thus preferred over direct matrix inversion. Note the power method for Eq. 7 is

$$R_{m+1} = AR_m + E, m = 0, 1, \cdots$$  \hspace{1cm} (8)$$

### 4 Web Graph Modeling

Before we start discussing the algorithm, we assume that the web graph is already available to us. However, this is generally not the case. We need a specific way of representing the web page and their interconnections and an algorithm to figure out the structure of the web. In this section, we will discuss those two problems.
4.1 Link Structure of the Web

Each web page is uniquely determined by its Uniform Resource Locator (URL), which is a specific character string that constitutes a reference to an Internet resource. Thus, we could associate each URL with an unique integer to represent the web page. Each hyperlink that appears on a web page indicates a directed edge linking from current page to the one identified with that hyperlink.

4.2 Web Crawling

Crawling is the process of collecting Web pages. I use Bread First Search (BFS) method to crawl the web and identify the structure of the networks. The algorithms is described as follows. Consider we begin with a single web page in the network, and assign it with a number 1, as shown in Fig. 5a. This page links to several pages and we are going to assign each page with a number as shown in Fig. 5b. From Fig. 5b, we observe that there is a link from page 2 to another page, thus we assign another number. Then we switch to page 3, assigning a number to the page linked by page 3, and so on. Fig. 5c gives us the crawled web structure represented by adjacency list.

By this example, we see that using the BFS algorithm we could identify the structure of the web. Generally, it’s not possible to crawl the whole web by only starting from one page. Therefore, in real applications, a group of source pages would be chosen to do large-scale web crawling which is out of the scope of the course project.
5 Implementation

The PageRank and web crawling algorithm are implemented in C++ for the project. The interface is designed using MFC (Microsoft Foundation Classes). The core data structures and algorithms will be discussed in this section.

5.1 Core Data Structure

The implementation uses adjacency list as the representation of graph. Two types of nodes: Edge and WebNode are created as structs.

The structure of Edge nodes is

```cpp
struct Edge {
    int end; // the other end of the hyperlink
    int cost; // cost of the edge
    Edge* next; // point at the next edge
};
```

The structure of the WebNode is

```cpp
struct WebNode {
    int index; // node number
    double prScore; // page rank score
    string url; // the url of the website
    Edge* nextEdge; // first edge in its edge list
};
```

5.2 Core Algorithms

The most important two algorithms in this project is the PageRank algorithm and the web crawl algorithm. The PageRank algorithm reads an adjacency list as input and output the computed PageRank score to a text file in descending order. The detailed algorithm is as follows.

```cpp
void computePageRank(vector<WebNode>& nodeList) {
    double* oldScore = new double[nodeList.size()];
    double* E = new double[nodeList.size()];
    double DAMPING_FACTOR = 0.85;
    for (unsigned int i = 0; i < nodeList.size(); i++) {
        oldScore[i] = 1.0 / double(nodeList.size()); // initialize each node with the same prScore
        E[i] = 0.15 / double(nodeList.size()); // initialize source of rank
    }
    double delta = 0.0, oldSum = 1.0;
    int loop = 0;
    do {
        // compute the page rank score
        for (unsigned int i = 0; i < nodeList.size(); i++) {
            int outDegree = 0;
            Edge* p = nodeList[i].nextEdge;
            while (p) {
                p = p->next;
                outDegree++;
            }
```
The web crawling algorithm reads an URL address as the starting web address and save crawled web structure information in a from-to URL text file. The detailed algorithm is as follows.

```cpp
void webCrawl(const char* startURL, char* fileName) {
    ofstream fout;
    fout.open(fileName);
    fout<<flush;
    fout.close();

    vector<string> urlList;
    vector<string> markedUrlList;
    queue<string> urlToVisit;
    markedUrlList.push_back(string(startURL));
    urlToVisit.push(string(startURL));

    int visitedUrlCnt = 0;
    while(urlToVisit.size() > 0 && visitedUrlCnt <= 15) {
        int outDegree = outDegree(urlToVisit.front());

        if (outDegree > 0) {
            p = nodeList[i].nextEdge;
            while (p) {
                nodeList[p->end].prScore += oldScore[i]/double(outDegree);
                p = p->next;
            }
        } else {
            for (unsigned int j = 0; j < nodeList.size(); j++) {
                nodeList[j].prScore += oldScore[i]/double(nodeList.size());
            }
        }

        for (unsigned int i = 0; i < nodeList.size(); i++) {
            nodeList[i].prScore = nodeList[i].prScore*DAMPING_FACTOR + (1-DAMPING_FACTOR)/double(nodeList.size());
        }

        // estimate the error between two consecutive iterations
        delta = 0.0;
        for (unsigned int i = 0; i < nodeList.size(); i++) {
            delta += abs(nodeList[i].prScore-oldScore[i]);
        }

        // prepare for the next iteration
        oldScore = nodeList.prScore;
        if (delta > 1e-4) {
            nodeList[i].prScore = 0.0;
        }
    }
    delete oldScore;
    delete E;
    delete sortedIndex;
}
```

```cpp
int outDegree(string startURL) {
    // score the link information in a text file
    ofstream fout;
    fout.open(fileName);
    fout<<flush;
    fout.close();

    // store the link information in a text file
    if (visitedUrlCnt == 0) {
        visitedUrlCnt = 15;
    }
}
```

The web crawling algorithm reads an URL address as the starting web address and save crawled web structure information in a from-to URL text file. The detailed algorithm is as follows.

```cpp
void webCrawl(const char* startURL, char* fileName) {
    ofstream fout;
    fout.open(fileName);
    fout<<flush;
    fout.close();

    vector<string> urlList;
    vector<string> markedUrlList;
    queue<string> urlToVisit;
    markedUrlList.push_back(string(startURL));
    urlToVisit.push(string(startURL));

    int visitedUrlCnt = 0;
    while(urlToVisit.size() > 0 && visitedUrlCnt <= 15) {
        int outDegree = outDegree(urlToVisit.front());

        if (outDegree > 0) {
            p = nodeList[i].nextEdge;
            while (p) {
                nodeList[p->end].prScore += oldScore[i]/double(outDegree);
                p = p->next;
            }
        } else {
            for (unsigned int j = 0; j < nodeList.size(); j++) {
                nodeList[j].prScore += oldScore[i]/double(nodeList.size());
            }
        }

        for (unsigned int i = 0; i < nodeList.size(); i++) {
            nodeList[i].prScore = nodeList[i].prScore*DAMPING_FACTOR + (1-DAMPING_FACTOR)/double(nodeList.size());
        }

        // estimate the error between two consecutive iterations
        delta = 0.0;
        for (unsigned int i = 0; i < nodeList.size(); i++) {
            delta += abs(nodeList[i].prScore-oldScore[i]);
        }

        // prepare for the next iteration
        oldScore = nodeList.prScore;
        if (delta > 1e-4) {
            nodeList[i].prScore = 0.0;
        }
    }
    delete oldScore;
    delete E;
    delete sortedIndex;
}
```
5.3 Function Specification

In this project, the primary functions and their usage are listed and explained as follows.

```cpp
void loadWebpageToStr(string& page, const char* url);
```

This function load the html content of the web page identified by `url` to a string for parsing and url extraction.

```cpp
void extractUrlFromStr(vector<string>& urlList, string str, const char* pUrl, const char* fileName);
```

This function parse the string `str` containing the html content of current web page (identified by `pUrl`), and extract the valid hyperlinks. For every valid link `lUrl` found on the page, a tuple (pUrl, lUrl) will be saved to the text file named `fileName`, which we call a from-to URL list file.

```cpp
bool searchUrl(vector<string> urlList, string str);
```

This function returns true if the string `str` is and element of the string vector `urlList`, and false otherwise.

```cpp
void extractUrlFromWebpage(vector<string>& urlList, const char* url, const char* fileName);
```

This function extract all the valid hyperlinks from a web page identified by `url`, save all of them into a string vector `urlList`, and writes a from-to URL list file named `fileName`.

```cpp
void webCrawl(const char* startURL, const char* fileName);
```
This function crawl a subgraph of the entire web graph starting from the web page identified by startURL, and save the link information into a from-to URL list named fileName.

```cpp
void readUrlText(vector<string>& urlList, const char* filename);
```

This function read the from-to URL list file named filename into a string vector urlList.

```cpp
void convertUrlListToGraph(const char* filename, vector<WebNode>& nodeList);
```

This function convert the from-to URL list file into a graph represented by an adjacency list.

```cpp
void sortPrScore(int* sortedIndex, vector<WebNode>& nodeList);
```

This function sort the web pages according to their PageRank score in descending order. It’s internally called by the function void computePageRank(vector<WebNode>&)

```cpp
void computePageRank(vector<WebNode>& nodeList);
```

The PageRank algorithm reads an adjacency list as input and output the computed PageRank score to a text file in descending order.

### 5.4 Data File Format

The program will generate four text files storing urls of all the web page crawled and PageRank results. The default file name for the four text files are "urllist.txt", "linklist.txt", "webgraph.txt", "pagerank.txt". The format of those files are specified as follows:

"urllist.txt" Each row in this file contains two columns: the first column is a integer number representing the index of the URL in the second column. A sample file is shown as follows:

<table>
<thead>
<tr>
<th>Index</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><a href="http://www.ucf.edu">http://www.ucf.edu</a></td>
</tr>
<tr>
<td>1</td>
<td><a href="https://my.ucf.edu/?promo_id=myUCF">https://my.ucf.edu/?promo_id=myUCF</a></td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.ucf.edu/academics">http://www.ucf.edu/academics</a></td>
</tr>
<tr>
<td>3</td>
<td><a href="http://www.ucf.edu/admissions">http://www.ucf.edu/admissions</a></td>
</tr>
<tr>
<td>4</td>
<td><a href="http://www.ucf.edu/research">http://www.ucf.edu/research</a></td>
</tr>
<tr>
<td>5</td>
<td><a href="http://www.ucf.edu/locations">http://www.ucf.edu/locations</a></td>
</tr>
<tr>
<td>6</td>
<td><a href="http://www.ucf.edu/campus_life">http://www.ucf.edu/campus_life</a></td>
</tr>
<tr>
<td>7</td>
<td><a href="http://www.ucf.edu/alumni_and_friends">http://www.ucf.edu/alumni_and_friends</a></td>
</tr>
<tr>
<td>8</td>
<td><a href="http://www.ucf.edu/athletics">http://www.ucf.edu/athletics</a></td>
</tr>
<tr>
<td>9</td>
<td><a href="http://www.ucf.edu/apply_to_ucf">http://www.ucf.edu/apply_to_ucf</a></td>
</tr>
</tbody>
</table>

"linklist.txt" Each row in this file corresponds to a directed edge in the graph, indicating the hyperlink towards the second page appears in the first web page. A sample file is shown as follows:
"webgraph.txt" This file stores the adjacency list representation of the built directed web graph. A sample file is as follows:

```
0->1->2->3->4
1->3->4
2->3->4
3->0->2
4->1
```

"pagerank.txt" This file stores the score of web page in descending order. A sample file is as follows:

<table>
<thead>
<tr>
<th>Node</th>
<th>PageRank</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0034034</td>
<td><a href="http://www.ucf.edu">http://www.ucf.edu</a></td>
</tr>
<tr>
<td>16</td>
<td>0.00260776</td>
<td><a href="http://today.ucf.edu">http://today.ucf.edu</a></td>
</tr>
<tr>
<td>385</td>
<td>0.00255723</td>
<td><a href="http://twitter.com/share">http://twitter.com/share</a></td>
</tr>
<tr>
<td>1</td>
<td>0.0025085</td>
<td><a href="https://my.ucf.edu/?promo_id=myUCF">https://my.ucf.edu/?promo_id=myUCF</a></td>
</tr>
<tr>
<td>29</td>
<td>0.00237236</td>
<td><a href="http://www.cah.ucf.edu">http://www.cah.ucf.edu</a></td>
</tr>
<tr>
<td>19</td>
<td>0.00236941</td>
<td><a href="http://map.ucf.edu/?show=2">http://map.ucf.edu/?show=2</a></td>
</tr>
<tr>
<td>33</td>
<td>0.00236539</td>
<td><a href="http://www.graduate.ucf.edu">http://www.graduate.ucf.edu</a></td>
</tr>
<tr>
<td>38</td>
<td>0.00233402</td>
<td><a href="http://www.nursing.ucf.edu">http://www.nursing.ucf.edu</a></td>
</tr>
<tr>
<td>30</td>
<td>0.00233086</td>
<td><a href="http://www.bus.ucf.edu">http://www.bus.ucf.edu</a></td>
</tr>
<tr>
<td>31</td>
<td>0.00233086</td>
<td><a href="http://education.ucf.edu">http://education.ucf.edu</a></td>
</tr>
</tbody>
</table>

5.5 Problems and Difficulties

The major difficulties encountered during the implementation is threefold. Firstly, I have to build a customized web crawler for this specific project, rather than using the existing commercial web crawler software from the internet, because the data format provided by those software are not intended for graph representation. Secondly, during the implementation of my own web crawler, the most difficult part is to parse each html for extraction of urls. The difficulty comes from the fact that hyperlinks appear on the web page in various forms. For example, some links are give as complete url address starting with "http://" or "https://", others, however, may start with relative address to the current web page. Moreover, some explicit url addresses contained in the html content may not be the hyperlinks on the web graph. And there exists infinitely large sites, pages and even URLs. And a fraction of the web pages have incorrect html, thus making the parser design a very difficult task. Thirdly, after the web graph has
been built by proper crawl, a lot of the memory space is dynamically allocated to the adjacency list representation of the graph. I have to be very careful with memory leak issues. I solved this problem by writing functions specifically dealing with memory allocation and reclaim.

5.6 Results

As result of the infinite size of the World Wide Web, it’s not possible for me to do such a huge-scale web crawl for this course project. Therefore, I choose to start from the UCF homepage “http://www.ucf.edu” and focus only on World Wide Web urls (i.e. urls starts with “http://www”). Even for this sub-network, the number of web sites is in the order tens of thousands. A sample of the result by a crawl of about 2000 web pages on December 1st, 2012 is shown in Table 1.

<table>
<thead>
<tr>
<th>Web Site</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ucf.edu">http://www.ucf.edu</a></td>
<td>0.00218</td>
</tr>
<tr>
<td><a href="http://www.housing.ucf.edu">http://www.housing.ucf.edu</a></td>
<td>0.00103</td>
</tr>
<tr>
<td><a href="http://www.graduate.ucf.edu">http://www.graduate.ucf.edu</a></td>
<td>0.00079</td>
</tr>
<tr>
<td><a href="http://www.cah.ucf.edu">http://www.cah.ucf.edu</a></td>
<td>0.00077</td>
</tr>
<tr>
<td><a href="http://www.cos.ucf.edu">http://www.cos.ucf.edu</a></td>
<td>0.00076</td>
</tr>
<tr>
<td><a href="http://www.med.ucf.edu">http://www.med.ucf.edu</a></td>
<td>0.00076</td>
</tr>
<tr>
<td><a href="http://www.sdes.ucf.edu">http://www.sdes.ucf.edu</a></td>
<td>0.00074</td>
</tr>
<tr>
<td><a href="http://www.bus.ucf.edu">http://www.bus.ucf.edu</a></td>
<td>0.00074</td>
</tr>
<tr>
<td><a href="http://www.rec.ucf.edu/im/a">http://www.rec.ucf.edu/im/a</a></td>
<td>0.00073</td>
</tr>
<tr>
<td><a href="http://www.creol.ucf.edu">http://www.creol.ucf.edu</a></td>
<td>0.00072</td>
</tr>
</tbody>
</table>

Table 1: A Sample PageRank of Top 10 Web Sites among 2000 WWW Web Sites Crawled Starting from UCF Homepage

From the sample results, it’s interesting to see that the home page of UCF ranks on the top, which is quite intuitive. It is easy to evaluate the PageRank results if the web graph under consideration is small enough. However, it becomes quite hard to confirm the results when it comes to large networks, and even so considering the web is changing constantly over time. The PageRank results we provide here can only be taken as a reference.

6 Conclusion

In this project, I presented PageRank algorithm, which is the original work by the two founders of Google, Lawrence Page and Sergey Brin. The algorithm is explained in detail and the special cases are individually considered. I implemented a web crawling algorithm to crawl the web as a directed graph represented with adjacency list. Based the adjacency list representation of the graph, I implemented the PageRank algorithm and showed some sample results by crawling the university web sites. The final program is wrapped up in an application with the interface designed with MFC.

A The Complete Algorithm Code
```
// Algorithm.h

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <iomanip>
#include <afxinet.h>
#include <queue>
#include <afxwin.h>
using namespace std;

struct AlgoPara {
    CEdit* dispWindow;
    CString startAddr;
    CString linkList;
    CString urlList;
    CString webGraph;
    CString include;
    CString exclude;
    int rLimit;
    double d;
    int preference;
};

struct Edge {
    Edge::Edge(int v, int c, Edge* e) { end = v; cost = c; next = e; }
    int end; // the other end of the hyper-link
    int cost; // cost of the edge
    Edge* next; // point at the next edge
};

struct WebNode {
    WebNode::WebNode(int i, double pr, string l, Edge* e) { index = i;
    prScore = pr; url = l; nextEdge = e; }
    int index; // node number
    double prScore; // page rank score
    string url; // the url of the website
    Edge* nextEdge; // first edge in its edge list
};

// read url list from text file
bool readUrlText(vector<string>& urlList, const char* filename);

// convert the url list into a graph represented by adjacency list
bool convertUrlListToGraph(const char* filename, vector<WebNode>& nodeList);

void sortPrScore(int* sortedIndex, vector<WebNode>& nodeList);

void computePageRank(vector<WebNode>& nodeList);

void extractUrlFromStr(vector<string>& urlList, string str, const char* pUrl, const char* fileName);

void loadWebpageToStr(string& page, const char* url);

void extractUrlFromWebpage(vector<string>& urlList, const char* url, const char* fileName);

void clearMemory(vector<WebNode>& nodeList);

bool searchUrl(vector<string> urlList, string str);
```
void webCrawl(const char* startURL, const char* fileName);

// Algorithm.cpp

#include "Algorithm.h"
extern AlgoPara parameter;

// read url list from text file
bool readUrlText(vector<string>& urlList, const char* filename) {
    ifstream dataFile(filename, ios_base::in);
    if (!dataFile) {
        cout<<"Error opening data file!"<<endl;
        return false;
    }
    string p;
    cout<<"**********************************************************************"<<endl<<"The list of urls:"
    while (!dataFile.eof()) {
        dataFile>>p;
        urlList.push_back(p);
        cout<<p<<endl;
    }
    dataFile.close();
    return true;
}

// convert the url list into a graph represented by adjacency list
bool convertUrlListToGraph(const char* filename, vector<WebNode>& nodeList) {
    Edge* e;
    vector<string> urlList;
    //vector<WebNode> nodeList;
    if (!readUrlText(urlList, filename)) {
        return false;
    }
    cout<<"**********************************************************************"<<endl<<"The list of web nodes:"
    nodeList.clear();
    for (unsigned int i = 0; i < urlList.size()/2; i ++) {
        string from = urlList[2*i]; // the source url address
        string to = urlList[2*i+1]; // the destination url address
        int fromIndex = -1;
        int toIndex = -1;
        // check if the from page already exists
        for (unsigned int j = 0; j < nodeList.size(); j++) {
            if (!from.compare(nodeList[j].url)) { // urls are the same
                fromIndex = j;
                break;
            }
        }
        if (fromIndex == -1) { // the source url does not exist or just created
            fromIndex = nodeList.size(); // will create a new node with fromIndex
            nodeList.push_back(WebNode(fromIndex, 0, from, NULL)); // create a new node
            cout<<"Node "<<fromIndex<<": "<<nodeList[fromIndex].url<<endl;
        }
        // check if the to page already exists
        for (unsigned int j = 0; j < nodeList.size(); j++) {
            if (!to.compare(nodeList[j].url)) { // urls are the same
                toIndex = j;
            }
        }
        if (toIndex == -1) { // the destination url does not exist or just created
            toIndex = nodeList.size(); // will create a new node with toIndex
            nodeList.push_back(WebNode(toIndex, 0, to, NULL)); // create a new node
            cout<<"Node "<<toIndex<<": "<<nodeList[toIndex].url<<endl;
        }
        nodeList[fromIndex].first->second[toIndex] = 1;
    }
}
toIndex = j;
break;
}
}
if (toIndex == -1) {
    // the destination url does not exist
    toIndex = nodeList.size();
    nodeList.push_back(WebNode(toIndex, 0, to, NULL));  // create a new node
    cout<<"Node ":<toIndex<<" : ":<nodeList[toIndex].url<<endl;
}
e = new Edge(toIndex, 0, NULL);  // create a new edge
Edge* p = nodeList[fromIndex].nextEdge;
if (p == NULL) {
    nodeList[fromIndex].nextEdge = e;
} else {
    while (p->next) {
        p = p->next;
    }
    p->next = e;
}
// debug
ofstream fout;
fout.open(parameter.webGraph);
cout<<"**********************************************************"<<endl<<"The adjacency list of web graph:"<<endl;
// parameter.dispWindow->ReplaceSel("**********************************************************\n\nThe adjacency list of web graph:\n\n"), for (unsigned int i = 0; i < nodeList.size(); i++) {
    cout<<nodeList[i].index;
    fout<<nodeList[i].index;
    CString str; str.Format("%d", nodeList[i].index);
    // parameter.dispWindow->ReplaceSel(str.GetBuffer());
    Edge* p = nodeList[i].nextEdge;
    while (p) {
        cout<<"->"<<p->end;
        fout<<"->"<<p->end;
        // parameter.dispWindow->ReplaceSel("->");
        str.Format("%d", p->end);
        // parameter.dispWindow->ReplaceSel(str.GetBuffer());
        p = p->next;
    }
    cout<<endl;
    fout<<endl;
    // parameter.dispWindow->ReplaceSel("\n");
}
fout<<flush;
fout.close();
return true;
}
void sortPrScore(int *sortedIndex, vector<WebNode>& nodeList) {
    ofstream fout;
    fout.open("output.txt");
    vector<WebNode> t(nodeList);
    cout<<"**********************************************************"<<endl<<"The page rank score :""<<endl;
    for (unsigned int i = 0; i < nodeList.size(); i++) {
        cout<"Node ":<i<<" Page Rank ":<nodeList[i].prScore<<endl;
    }
}
cout << "**********************************************************
The sorted page rank score :
**********************************************************
The top 10 web pages :
for (unsigned int i = 0; i < nodeList.size(); i++) {
    sortedIndex[i] = t[0].index;
    double maxScore = t[0].prScore;
    int maxIndex = 0;
    for (unsigned int j = 1; j < t.size(); j++) {
        if (maxScore < t[j].prScore) {
            maxScore = t[j].prScore;
            sortedIndex[i] = t[j].index;
            maxIndex = j;
        }
    }
    t.erase(t.begin()+maxIndex);
    // debug
    cout << "Node " << sortedIndex[i] << " PageRank " << nodeList[sortedIndex[i]].prScore << " <" <nodeList[sortedIndex[i]].url << endl;
    fout << "Node " << sortedIndex[i] << " PageRank " << nodeList[sortedIndex[i]].prScore << " <" <nodeList[sortedIndex[i]].url << endl;
    // UI
    if (i < 10) {
        // parameter.dispWindow->ReplaceSel("Node ");
        CString str; str.Format("%d", sortedIndex[i]);
        parameter.dispWindow->ReplaceSel(" PageRank ");
        str.Format("%.5f", nodeList[sortedIndex[i]].prScore);
        parameter.dispWindow->ReplaceSel(str);
        parameter.dispWindow->ReplaceSel(" ");
        parameter.dispWindow->ReplaceSel(nodeList[sortedIndex[i]].url_c_str());
        parameter.dispWindow->ReplaceSel(" ");
    }
}
fout << flush;
fout.close();
}
void computePageRank(vector<WebNode>& nodeList) {
double* oldScore = new double[nodeList.size()];
double* E = new double[nodeList.size()];
double DAMPING_FACTOR = 0.85;
for (unsigned int i = 0; i < nodeList.size(); i++) {
    oldScore[i] = 1/double(nodeList.size()); // initialize each node with the same prScore
    E[i] = 0.15/double(nodeList.size()); // initialize source of rank
}
double delta = 0.0, oldSum = 1.0;
int loop = 0;
// debug
<<"**********************************************************
The page rank score :"<<endl;
do { // debug
    cout << "iter " << loop << "": ";
    for (unsigned int i = 0; i < nodeList.size(); i++) {
        cout << fixed << setprecision(4) << oldScore[i] " \n;
```cpp
} 
cout<<endl;
*/
// compute the page rank score
for (unsigned int i = 0; i < nodeList.size(); i++) {
    int outDegree = 0;
    Edge* p = nodeList[i].nextEdge;
    while (p) {
        p = p->next;
        outDegree++;
    }
    if (outDegree > 0) {
        p = nodeList[i].nextEdge;
        while (p) {
            nodeList[p->end].prScore += oldScore[i]/double(outDegree);
            p = p->next;
        }
    } else {
        for (unsigned int j = 0; j < nodeList.size(); j++) {
            nodeList[j].prScore += oldScore[i]/double(nodeList.size());
        }
    }
}
// estimate the error between two consecutive iterations
delta = 0.0;
for (unsigned int i = 0; i < nodeList.size(); i++) {
    delta += abs(nodeList[i].prScore-oldScore[i]);
}
// prepare for the next iteration
for (unsigned int i = 0; i < nodeList.size(); i++) {
    oldScore[i] = nodeList[i].prScore;
    if (delta > 1e-4) {
        nodeList[i].prScore = 0.0;
    }
}
while (delta > 1e-4);

int* sortedIndex = new int[nodeList.size()];
sortPrScore(sortedIndex, nodeList);
delete oldScore;
delete E;
delete sortedIndex;

void extractUrlFromStr(vector<string>& urlList, string str, const char* pUrl, const char* fileName) {
    int _state = 0;
    const int S_lar = 1, S_a = 2, S_sp = 3, S_rar = 4, S_h1 = 5, S_r = 6, S_e = 7, S_f = 8, S_eq = 9, S_quo = 10, S_h2 = 11, S_t1 =
```
const char* legal_chars = "abcdefghijklmnopqrstuvwxyz0123456789./\˜#%&;()_+-=;?
char *p, *mark;
char* s = new char[str.size()+1]; s[str.size()] = '\0';
strncpy(s, str.c_str(), str.size());
_state = 0;
//cout<<endl
<<"**********************************************************"<<endl<<"The extracted urls:"
for (p = s; *p ; p++) {
    switch(_state) {
    case 0:
        if(*p == '<') {
            _state = S_lar;
            mark = p;
        }
        break;
    case S_lar:
        if (*p == 'a') {
            _state = S_a;
        } else if (*p == ')') {
            _state = S_sp;
        } else {
            _state = 0;
        }
        break;
    case S_a:
        if (*p == '>') {
            _state = S_sp;
        } else {
            _state = 0;
        }
        break;
    case S_sp:
        if (*p == 'h') {
            _state = S_h1;
        } else if (*p == '>') {
            _state = 0;
        } else {
            _state = S_sp;
        }
        break;
    case S_h1:
        if (*p == 'r') {
            _state = S_r;
        } else if (*p == ')') {
            _state = S_sp;
        } else if (*p == '>'){
            _state = 0;
        } else {
            _state = S_a;
        }
        break;
    case S_r:
        if (*p == 'e') {
            _state = S_e;
        } else if (*p == ')') {
            _state = S_sp;
        } else if (*p == '>'){
            _state = 0;
else {
    _state = S_a;
}
break;
case S_e:
    if (*p == 'f') {
        _state = S_f;
    } else if (*p == 'r') {
        _state = S_sp;
    } else if (*p == '>') {
        _state = 0;
    } else {
        _state = S_a;
    }
    break;
case S_f:
    if (*p == '=') {
        _state = S_eq;
    } else if (*p == 'r') {
        _state = S_f;
    } else if (*p == '>') {
        _state = 0;
    } else {
        _state = S_a;
    }
    break;
case S_eq:
    if (*p == '"') {
        _state = S_quo;
    } else if (*p == 'r') {
        _state = S_eq;
    } break;
case S_quo:
    if(*p == 'h') {
        _state = S_h2;
        mark = p;
    } else if (*p == 'r') {
        _state = S_quo;
    } else if (*p == '/') {
        _state = S_sls2; // the url is written as relative url address
        mark = p;
    } break;
case S_h2:
    if(*p == 't') {
        _state = S_t1;
    } else {
        _state = 0;
    }
    break;
case S_t1:
    if(*p == 't') {
        _state = S_t2;
    } else {
        _state = 0;
    }
    break;
case S_t2:
    if(*p == 'p') {
        _state = S_p;
    }
else {
    _state = 0;
}
break;
case S_p:
    if (*p == ':') {
        _state = S_col;
    } else if (*p == 's') {
        _state = S_s;
    } else {
        _state = 0;
    }
    break;
case S_s:
    if (*p == ':') {
        _state = S_col;
    } else {
        _state = 0;
    }
    break;
case S_col:
    if (*p == '/') {
        _state = S_sls1;
    } else {
        _state = 0;
    }
    break;
case S_sls1:
    if (*p == '/') {
        _state = S_sls2;
    } else {
        _state = 0;
    }
    break;
case S_sls2:
    if (strchr(legal_chars, tolower(*p)) == NULL) {
        char* str = new char[p-mark+1]; str[p-mark] = '\0';
        strncpy(str, mark, p-mark);
        string url(str);
        if (url[0] == '/') { // relative path
            url.insert(0, pUrl);
        }
        if (url[url.size()-1] == '/') {
            url.erase(url.size()-1, 1);
        }
    } // filter
    if (parameter.include.GetLength() > 0) {
        if (url.find(parameter.include.GetBuffer()) != -1) {
            if (parameter.exclude.GetLength() > 0) {
                if (url.find(parameter.exclude.GetBuffer()) == -1) {
                    urlList.push_back(url);
                }
            } else {
                urlList.push_back(url);
            }
        }
    } else {
        urlList.push_back(url);
    }
} else {
    if (parameter.exclude.GetLength() > 0) {
if (url.find(parameter.exclude.GetBuffer()) == -1) {
    urlList.push_back(url);
} else {
    urlList.push_back(url);
}

// debug
// cout<<"url<<endl;
_state=0;
p--;  // backtrack
delete str;
}

if (_state) {
    char* str = new char[p-mark+1]; str[p-mark] = '\0';
    strncpy(str, mark, p-mark);
    string url(str);
    if (url[url.size()-1] == '/') {
        url.erase(url.size()-1, 1);
    }
    urlList.push_back(url);
    // debug
    // cout<<"url<<endl;
delete str;
} delete s;

void loadWebpageToStr(string& page, const char* url) {
    CInternetSession session("HttpClient");
    CHttpFile* pfile = (CHttpFile *)session.OpenURL(url);
    DWORD dwStatusCode;
    pfile -> QueryInfoStatusCode(dwStatusCode);
    CString content;
    // cout
    <<"**********************************************************"<<endl;
    if (dwStatusCode == HTTP_STATUS_OK) {
        CString data;
        while (pfile -> ReadString(data)) {
            content += data + "
";
        }
        content.TrimRight();
        // printf(" %s\n ",(LPCTSTR)content);
    }
    pfile -> Close();
delete pfile;
    session.Close();
    page += content.GetBuffer(content.GetLength());
}

void extractUrlFromWebpage(vector<string>& urlList, const char* url, const char* fileName) {
    string page;
    loadWebpageToStr(page, url);
    extractUrlFromStr(urlList, page, url, fileName);
    // put extracted url link into a from-to file
    ofstream fout;
    fout.open(fileName, ios_base::app);
for (unsigned int i = 0; i < urlList.size(); i++) {
    fout<<url<<" ";
    fout<<urlList[i].c_str()<<endl;
}        
fout<<flush;
fout.close();
}

void clearMemory(vector<WebNode>& nodeList) {
    Edge *current, *p;
    for (unsigned int i = 0; i < nodeList.size(); i++) {
        p = nodeList[i].nextEdge;
        while (p != NULL) {
            current = p->next;
            delete p;
            p = current;
        }
    }
}

bool searchUrl(vector<string> urlList, string str) {
    bool flag = false;
    for (unsigned int i = 0; i < urlList.size(); i++) {
        if (!urlList[i].compare(str)) {
            flag = true;
            break;
        }
    }
    return flag;
}

void webCrawl(const char* startURL, const char* fileName) {
    // store the link information in a text file
    ofstream fout;
    fout.open(fileName);
    fout<<flush;
    fout.close();

    vector<string> urlList;
    vector<string> markedUrlList;
    queue<string> urlToVisit;

    markedUrlList.push_back(string(startURL));
    urlToVisit.push(string(startURL)); // initialize the queue with the
    start url
    int visitedUrlCnt = 0;
    int listedUrlCnt = 0;

    while (urlToVisit.size() > 0 && listedUrlCnt <= parameter.rLimit) {
        // there are unvisited urls
        string root = urlToVisit.front(); // get the first url as the new root
        urlToVisit.pop(); // pop out visited url
        extractUrlFromWebpage(urlList, root.c_str(), fileName);
        cout<<++visitedUrlCnt<<" web page visited "<<root.c_str()<<" 
        
        "<<markedUrlList.size()<<" listed"<<endl;
        // UI
        parameter.dispWindow->ReplaceSel("Visiting:");
        parameter.dispWindow->ReplaceSel(root.c_str());
        CString str; str.Format("%d", markedUrlList.size());
        parameter.dispWindow->ReplaceSel(str);
        parameter.dispWindow->ReplaceSel(" listed\n");

        for (unsigned int i = 0; i < urlList.size(); i++) {
            if (!searchUrl(markedUrlList, urlList[i])) {
                markedUrlList.push_back(urlList[i]);
            }
        }
    }
}
urlToList.push(urlList[i]);
++listedUrlCnt;
}
}
urlList.clear();

// debug
// map each web-page to an integer and store the mapping in a text file
ofstream fout;
fout.open("webPageList.txt");
for (unsigned int i = 0; i < markedUrlList.size(); i++) {
fout<<i<<" "<<markedUrlList[i].c_str()<<endl;
// UI
//parameter.dispWindow->ReplaceSel(" ");
//parameter.dispWindow->ReplaceSel(markedUrlList[i].c_str());
//parameter.dispWindow->ReplaceSel("\r\n");
}
fout<<flush;
fout.close();

/*
int _tmain(int argc, char* argv[])
{
char* fileName = "linkList.txt";
vector<WebNode> nodeList; // store the graph as an adjacency list
// crawl the web from a starting url
webCrawl("http://www.ucf.edu", fileName);
// model the web as a graph represented in adjacency list
convertUrlListToGraph(fileName, nodeList);
// compute page rank
computePageRank(nodeList);
// clear memory
clearMemory(nodeList);
system("PAUSE");
return 0;
}
*/

References


[10] 187-page study from Graz University, Austria, includes the note that also human brains are used when determining the page rank in Google


