Accelerated Pagerank Algorithm

20337025 Canming Cui and 20337138 Zhanhao Xu

Department of Computer Science, Sun Yat-Sen University.

Abstract

This article is the final major assignment for the Principles and Techniques of Big Data class. We gave a brief overview of the PageRank algorithm and performed some optimizations and implemented three techniques for accelerating convergence, namely initial value design, Quadratic Extrapolation, and fix converged nodes. We give mathematical derivations and show in detail the experimental results in my report, which demonstrate the correctness and efficiency of the optimisation algorithm. In addition, We have implemented the PageRank algorithm in MapReduce form.

Keywords: PageRank, Quadratic Extrapolation, fix converged nodes, MapReduce

1 Introduction

The PageRank algorithm, developed by Larry Page and Sergey Brin[1], is used to evaluate and rank web pages based on their importance. It plays a vital role in search engine optimization and web page ranking.

The algorithm assesses the importance of web pages by considering the quantity and quality of links. Each web page is treated as a node, and links are viewed as directed edges between nodes(as shown in Fig. 1). By analyzing these links, the algorithm determines the significance of web pages.The core idea behind the algorithm is "link voting." When a web page is linked by other pages, it receives more "link votes," increasing its importance. The importance of a page is also influenced by the importance of the pages that link to it.

To calculate the importance of web pages, the PageRank algorithm uses an iterative approach. In each iteration, the algorithm redistributes link weights and performs calculations based on the updated weights. This process continues until convergence is achieved. The final result is a score assigned to each web page, reflecting its importance for ranking purposes.

Apart from search engine rankings, the PageRank algorithm finds applications in social network analysis and recommendation systems. It provides an effective way to evaluate node importance in networks and has significantly impacted information retrieval and network analysis

There have been many improvements to the PageRank algorithm over the last few decades. Our accelerated algorithm includes the Quadratic Extrapolation algorithm. It was proposed by Kamvar

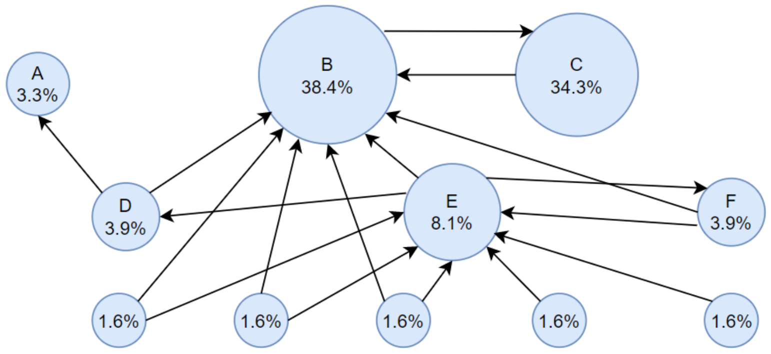

Figure 1: A simple illustration of the Pagerank algorithm. The percentage shows the perceived importance, and the arrows represent hyperlinks.

et al.[2], which aimed to expedite the convergence rate of the PageRank vector. Through their work, they achieved a significant improvement in convergence, reducing the convergence rate from 25% to 300%. W.Xing et al.[3] proposed and developed the weighted PageRank algorithm (WPR) as an extension to the standard PageRank algorithm. They introduced the concept of incorporating the importance of both inlinks and outlinks of web pages into the ranking process. Ispen and Selee[4] presented a fundamental algorithm for computing the PageRank vector using the Google matrix. They introduced a novel approach where all dangling nodes are combined into a single node. The algorithm calculates the PageRank of non-dangling nodes separately and then calculates the PageRank for the dangling nodes.

The innovation of our accelerated Pagerank algorithm is the combination of various techniques: sparse matrices, initial value design, Quadratic Extrapolation and fix converged nodes. And we combine it with MapReduce.

2 PageRank Review

The PageRank algorithm simulates user click behavior. Suppose there are users browsing web pages, and at a given moment, they randomly click on a link on the current page and navigate to another page. As time goes on, the number of users on each page should stabilize, representing the page's PageRank value.

Let num_nodes represents the total number of nodes, $Out[j]$ represents the number of outLinks of page j, $In[j]$ represents the number of inLinks of page j, $\beta(j)$ is the set of pages pointing into page j, $PageRank[j]$ represents the PageRank values of page j, $A[\ ][\ ]$ represents the adjacency matrix of pages, p represents the probability of random walk, and ϵ represents the convergence threshold. The formula for the aforementioned PageRank algorithm is expressed as **??**:

$$PageRank[i] = \sum_{j \in \beta(j)} \frac{PageRank[j]}{Out[j]} \tag{1}$$

To give precise and manageable meaning to such a roughly stated definition let us denote by A the adjacency matrix with the elements:

$$A[j][i] = a_{ij} = \begin{cases} \frac{1}{Out[j]} & \text{if } j \in \beta(i) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

It can be shown that each PageRank value converges (i.e. the magnitude of change tends to 0) when the adjacency matrix A between pages satisfies these three conditions, independent of the initial value: (i)A is a Markov matrix. (ii)A corresponds to a graph that is strongly connected. (iii)A is non-periodic. However, this is only the ideal case; in real web pages, the matrix A will not usually satisfy the above conditions. Therefore, to ensure that the algorithm converges correctly, we need to modify the matrix A slightly.

A Markov matrix is that all lements are greater than or equal to 0 and the sum of the elements of any row is equal to 1. Why we need A to be a Markov matrix? Becaues the equivalent condition for matrix A to be a Markov matrix is that all nodes have outLinks. If page i has no outLinks, meaning the i-th row of matrix A is all zeros, then the PageRank values that transition to this page will be lost and cannot be transferred to other pages. As a result, the sum of PageRank values in the entire system will continuously decrease until it reaches 0. Finally the PageRank values of all pages converge to 0. So we need to make matrix A to be a Markov matrix, i.e. we need to ensure that all pages without outLinks have outLinks. So, link the pages without outLinks to all pages:

$$A[j][i] = a_{ij} = \begin{cases} \frac{1}{Out[j]} & \text{if } j \in \beta(i) \\ \frac{1}{num\_nodes} & \text{if page j hasn't outLinks} \end{cases} \tag{3}$$

Strong connectivity ensures that every node can be reached from any other node through directed edges. Periodicity refers to the property of a matrix where the result of matrix multiplication repeats itself after a certain number of multiplications. It implies that the matrix follows a cyclic pattern in its multiplication process. If the matrix A does not satisfy strong connectivity, the value of PageRank at convergence can only be 0 or 1. If the matrix A does not satisfy non-periodicity, it may never converge.

A random wandering strategy is used to solve both problems, i.e. the user closes the current page and a new page is opened at random:

$$A[j][i] = (1 - p) \cdot A[j][i] + p \tag{4}$$

The equation for the transfer from the (k-1)th PageRank to the kth PageRank can be expressed as follows(by matrix multiplication):

$$PageRank_k = A \cdot PageRank_{k-1} \tag{5}$$

The mainstream PageRank algorithm primarily employs iterative calculation methods to quickly obtain the approximate convergence value of the PageRank, so the complexity of the PageRank algorithm lies in iterative computation. Next we will discuss iterative methods and acceleration.

3 Accelerated Algorithm

In this section we describe in detail the four accelerated iteration algorithms of the PageRank algorithm.

3.1 Sparse Matrix

The above mentioned PageRank algorithm based on matrix power implementation is as follows. Suppose we have obtained the adjacency matrix A from Eq. (3) and Eq. (4), and then initialise the values of PageRank, and the sum of PageRank is 1. Then start iterating by letting A multiply by PageRank until PageRank converges(the amount of change is less than threshold ϵ), and the algorithm would be if sparse matrices are not used:

Algorithm 1: Computing PageRank by the power method

```
An adjacency matrix $A$ has been obtained;
for all page $i$ do
    $PageRank_0[i] = \frac{1}{num\_nodes}$;
end
k=1;
while $\delta < \epsilon$ do
    $PageRank_k = A \cdot PageRank_{k-1}$;
    $\delta = max_i(\mid PageRank_k[i] - PageRank_{k-1}[i] \mid)$;
    $k+ = 1$;
end
```

The complexity of the algorithm is $O(num\_nodes^2 \cdot n)$, where n is the number of iterations. Since each element of the matrix A is non-zero, the multiplication time occupies most of the runtime. However, we can observe that most of the elements in A are the same, so they can be extracted as constants, and removing these elements from A will turn it into a sparse matrix. Sparse matrix can reduce the number of multiplications significantly compared to the original matrix. For elements in the matrix that are obtained by random wandering or no outLinks equally divided among all pages, they can be extracted as constants. Modify the update formula of PageRank as follows(where $sum_{k-1} = \sum_{\text{i hasn't outLinks}} PageRank_{k-1}[i]$):

$$PageRank_k = A' \cdot PageRank_{k-1} + sum_{k-1} \cdot (1-p) + p \tag{6}$$

The sparse matrix power iteration algorithm is as Algorithm 2, where $A'[i][j]$(j¡number of inLinks for page i) denotes the page from which the jth inLink of page i originates. The complexity of the algorithm can be analysed as $O(n \cdot (num\_nodes + e))$(e is the total number of links). Because in the sparse matrix:$num\_nodes + e < num\_nodes^2$, so the sparse matrix power algorithm complexity is lower than the original matrix power algorithm.

Algorithm 2: Computing PageRank by the power method using sparse matrix

```
An sparse matrix $A'$ has been obtained;
for all page $i$ do
    $PageRank_0[i] = \frac{1}{num\_nodes}$;
end
k=1;
while $\delta < \epsilon$ do
    $sum = 0$;
    for page $i$ that hasm't outLinks do
        $sum = sum + PageRank_{k-1}[i]$;
    end
    for all page $i$ do
        for $1 \leq j \leq In[i]$ do
            $PageRank_k[i] += PageRank_{k-1}[\frac{A'[i][j]\cdot(1-p)}{Out[[A'[i][k]]]}]$;
            $PageRank_k[i] += \frac{sum(1-p)}{num\_nodes}$;
            $PageRank_k[i] += p$;
        end
    end
    $\delta = max_i(\mid PageRank_k[i] - PageRank_{k-1}[i] \mid)$;
    $k += 1$;
end
```

3.2 Initial Value Design

The initial PageRank value can be equally distributed or determined based on some heuristic. However, the algorithm will converge faster if the initial value can be closer to the final converged state. So we set the initial value of PageRank according to the number of inLinks, and the size of the initial value is proportional to the number of inLinks, which is shown in Eq. (7)(*num_nodes* is added to prevent the denominator from being 0).

$$PageRank_0[i] = \frac{In[i] + 1}{\sum_{\leq j \leq num\_nodes} In[j] + num\_nodes} \tag{7}$$

The code implementation is as follows, see Appendix A for full codes:

```
//init the value of matrix
void Init_value(int Init){
        if(Init==0){
                for(int i=1;i<=num_total;i++){
                        val[i]=1/double(num_total);
                }
        }
        if(Init==1){
                int inlinks_sum=0;
                for(int i=1;i<=num_total;i++){
                        inlinks_sum+=count_inlinks[i]+1 ;
                }
                for(int i=1;i<=num_total;i++){
                        val[i]=(count_inlinks[i]+1)/double(inlinks_sum);
                }
        }
}
```

3.3 Quadratic Extrapolation

The quadratic extrapolation method comes from the work of Kamvar et al[2]. and we have applied it based on the guidance of their paper. The quadratic extrapolation method is a technique that speeds up the iterative process of the PageRank algorithm. It leverages the convergence properties of PageRank values. This method assumes that the change in PageRank values follows a quadratic convergence pattern, meaning that the change in PageRank is proportional to the square of the iteration step. Based on this assumption, the quadratic extrapolation method can use the results of the early iterations to estimate the values of the subsequent iterations, thereby accelerating the convergence of the algorithm.

The steps of the quadratic extrapolation method are as follows:

1. Assume that $PageRank_{k-2}$ is a linear combination of the feature vectors corresponding to the first and second features:

$$PageRank_{k-2} = u_1 + \alpha_2 u_2$$

2. From the mathematical analysis we have that the maximum eigenvalue $\lambda = 1$, so for k-1 iterations and k iterations :

$$PageRank_{k-1} = A \cdot PageRank_{k-2} = u_1 + \alpha_2 \lambda_2 u_2$$

$$PageRank_k = A \cdot PageRank_{k-1} = u_1 + \alpha_2 \lambda_2^2 u_2$$

3. Let the variables g,h:

$$g = PageRank_{k-1} - PageRank_{k-2} = \alpha_2(\lambda_2 - 1)u_2$$

$$h = PageRank_k - PageRank_{k-1} = \alpha_2(\lambda_2 - 1)\lambda_2 u_2$$

4. The formula for calculating λ_2 is:

$$\lambda_2 = \frac{h^T \cdot g}{g^T \cdot g}$$

5. Convergence can be accelerated by updating the original PageRank with the new value of

PageRank as follows:

$$PageRank_k = u_1 = \frac{\lambda_2 PR_{k-1} - PR_k}{\lambda_2 - 1}$$

The code implementation is as follows, see Appendix A for full codes:

```
//Quadratic Extrapolation method
void Quadratic_Extrapolation(){
        double lamda;
        double sum1=0;
        double sum2=0;
        //Calculated as shown in the formula
        for(int i=1;i<=num_total;i++){
                sum1+=last_difference[i]*last_difference[i];
                sum2+=last_difference[i]*dif[i];
        }
        lamda=sum2/sum1;
        for(int i=1;i<=num_total;i++){
                newval[i]=(lamda*val[i]-newval[i])/(lamda-1);
        }
}
```

3.4 Fix Converged nodes

In the iterative calculation of PageRank algorithm, the convergence speed of different nodes is different, some nodes converge faster, but because other nodes have not converged, then the converging nodes have to follow along with the iterative calculation, which will cost a lot of unnecessary calculations. Therefore, we can fix the value of the converged nodes to reduce the amount of computation in the iterative process.

We can monitor δ(The amount of change in PageRank for the two iterations before and after) of a node during the iteration process, and if it is less than a certain threshold, it will be judged to have converged. And this threshold needs many experiments to select the best threshold.

The code implementation is as follows, see Appendix A for full codes:

```
...
...
for (int iter=1;iter<max_iteration;iter++){
        double sum_val0=0;
        // Calculate the sum of values for nodes with no outgoing links
        for (int j=0;j<num_of_nooutlinks;j++)
        sum_val0+=val[no_outlinks[j]];
        // Calculate the new PageRank values for each node
        for (int i=1;i<=num_total;i++){
            if(Fix&&is_Converge[i])continue;
            // Nodes with outgoing links
            // If the node has outgoing links, assign it a value based
                on the sum of values for nodes with no outgoing links
            if(count_outlinks[i]!=0)newval[i]=Poblity/double(num_total)
                +(1-Poblity)*sum_val0/double(num_total-1);
            // If the node has no outgoing links, assign it a value
                based on the sum of values for nodes with no outgoing
                links
            else newval[i]=Poblity/double(num_total)+(1-Poblity)*(
                sum_val0-val[i])/double(num_total-1);
            // The second part of the PageRank value is obtained by "
                absorbing" from all incoming edges to the node
            for(int j=0;j<count_inlinks[i];j++){
                // The new PageRank value of node i is updated by adding
                     the probability of not performing a random walk,
                // the weight of the j-th incoming edge of node i, and
                    the PageRank value of the incoming node
                newval[i]+=(1-Poblity)*val[spares_matrix_cite[i][j]]*
                    weight[i][j];
            }
        }
        ...
        ...
}
...
...
```

4 MapReduce

The idea for the section comes from the work of ST Wierzchoń et al[5]. The MapReduce programming framework was proposed by Dean and Ghemawat[6]. It is used for handling and producing vast amounts of data. It involves two main functions: the map function and the reduce function. The map function takes a key/value pair and generates a collection of intermediate key/value pairs. The reduce function then combines all the intermediate values associated with the same intermediate key.

The process can be summarized as follows:

1. Data Splitting: The input dataset is divided into multiple equally sized data blocks, which are processed in parallel.
2. Mapping: Each data block is processed by a set of mappers, applying a user-defined map function to produce intermediate key-value pairs.

3. Shuffling and sorting: Intermediate key-value pairs are sorted and grouped by keys to aggregate values with the same key.

4. Reducing: Each reducer processes intermediate keys and their associated values using a user-defined reduce function.

5. Output generation: Results from reducers are written to output files or storage systems.

The "mapper" and "reducer" can be represented symbolically in the following manner($< k1, v1 >$ represents a key-value pair, where 'k1' is the key and 'v1' is the corresponding value. Additionally, the use of $[\cdot]$ denotes a list containing multiple elements):

$$\text{map:} \; < k_1, v_1 > \to [< k_2, v_2 >]$$

$$\text{reduce:} \; < k\_2, [v_2] > \to [< k_3, v_3 >]$$

In order to implement Algorithm 1 within the MapReduce framework, a dual set of map-reduce functions needs to be applied. The first pair of map-reduce functions is utilized to perform matrix-vector multiplication. First, knowing page i, its current PageRank $PageRank[i]$, and the list of outlinks $OutLinks_i$, the mapper returns the set of pairs $< j, w_j >$, where j is a page linked from i and $w_j = PageRank[i]/out_i$, with out_i being the length of $OutLinks_i$. The reducer is responsible for aggregating the information:

$$\text{map:} \; < i; (PageRank[i], OutLinks_i) > \to [< j, PageRank[i]/out_i >]$$

$$\text{reduce:} \; < j, [w_i] > \to [< j, \sum_i w_i >]$$

The second pair finishes the job, i.e. the pairs $< j, \sum_i w_i >$ are transmitted by the mapper to the reducer that computes their sum. We can also implement the optimised algorithm by a similar way.

5 Experiment

5.1 Data Set

We chose the Wikipedia voting dataset for our experiments(https://snap.stanford.edu/data/wiki-Vote.html). The network contains all the Wikipedia voting data from the inception of Wikipedia till January 2008. Nodes in the network represent wikipedia users and a directed edge from node i to node j represents that user i voted on user j. The network has 7115 nodes and 103689 directed edges.

5.2 Results

In order to compare the performance of various versions of the PageRank algorithm, we compare the execution time of each version of the algorithm to reach the convergence threshold in terms of the order of magnitude of the convergence threshold, and the resulting table is shown in Tab. 1.

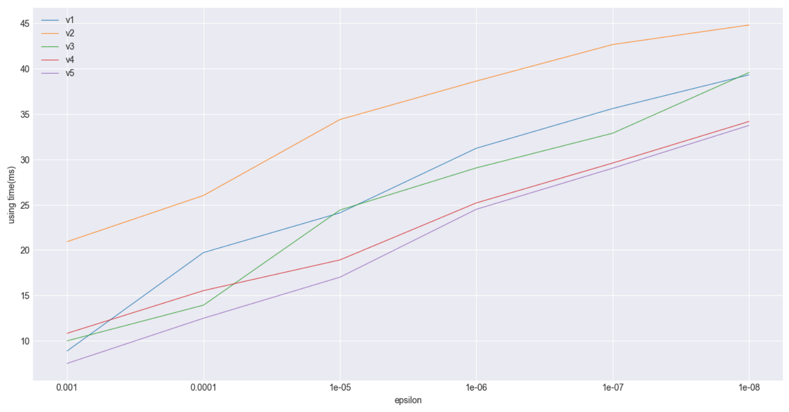

Figure 2: Line graph of execution time for different versions.

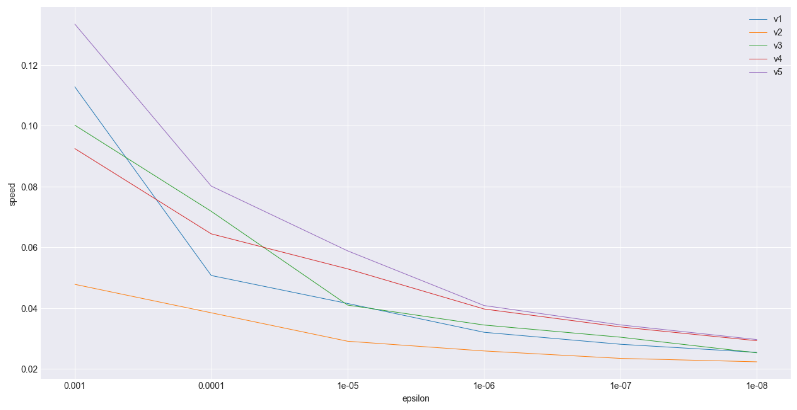

Figure 3: Line graph of speed for different versions.

The meaning of the symbols in the table is as follows:

- origin: The basic Pagerank implemented with a dense adjacency matrix.
- v_1: Sparse matrix optimisation.
- v_2: Sparse matrix + Initial value design.
- v_3: Sparse matrix + Quadratic Extrapolation.
- v_4: Sparse matrix + Fix Converged nodes
- v_5: Accelerated Pagerank (Sparse matrix + Quadratic Extrapolation + Fix Converged nodes)

In order to show the experimental results more intuitively, they are plotted as a line graph, as shown in Fig. 2 and Fig. 3(the pagerank of dense matrices takes longer and is omitted here).

ϵ	*origin*	v_1	v_2	v_3	v_4	v_5
10^{-3}	144.555	8.867	20.917	9.983	10.814	7.495
10^{-4}	319.118	19.717	26.011	13.921	15.522	12.478
10^{-5}	461.762	24.092	34.380	24.401	18.905	17.000
10^{-6}	614.731	31.214	38.623	29.051	25.189	24.486
10^{-7}	791.330	35.597	42.651	32.875	29.588	29.016
10^{-8}	945.651	39.312	44.803	39.572	34.172	33.737

Table 1: Table of execution times for different convergence thresholds, table entry represents execution time (ms)

From the experimental results, it can be learnt that the execution time of all algorithms rise monotonically with decreasing threshold ϵ, which is approximately linear. Each optimisation plays a role, i.e., speeds up the execution of the Pagerank algorithm. And the Accelerated PageRank algorithm has the best performance among all versions, which proves that our accelerated algorithm is effective.

6 Summary

In this article, we give a brief overview of the PageRank algorithm and improve it. In Sec. 1, we introduced some concept, background and related works of the PageRank algorithm. In Sec. 2, we described the principles of the PageRank algorithm. In Sec. 3, we present the accelerated PageRank algorithm, applying four optimisations. In Sec. 4 we described the MapReduce framework and how to apply it to PageRank algorithm. In Sec. 5, we show and analyse the results of the experiments, which demonstrate the effectiveness of the accelerated algorithm. In the end, we learnt a lot through this final assignment and our abilities were honed.

References

[1] Sergey Brin and Lawrence Page. “The anatomy of a large-scale hypertextual web search engine”. In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[2] Sepandar D Kamvar et al. “Extrapolation methods for accelerating PageRank computations”. In: *Proceedings of the 12th international conference on World Wide Web.* 2003, pp. 261–270.

[3] Wenpu Xing and Ali Ghorbani. “Weighted pagerank algorithm”. In: *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.* IEEE. 2004, pp. 305–314.

[4] Ilse CF Ipsen and Teresa M Selee. “PageRank computation, with special attention to dangling nodes”. In: *SIAM Journal on Matrix Analysis and Applications* 29.4 (2008), pp. 1281–1296.

[5] Sławomir T Wierzchoń et al. “Accelerating PageRank computations”. In: *Control and Cybernetics* 40.2 (2011), pp. 259–274.

[6] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

A Appendix

Run the main function of the programme for conducting experiments.

main.c:

```
//main function
int main(int argc, char *argv[]) {
    bool N=0;
        bool I=0;
        bool E=0;
        bool F=0;
        bool W=0;
        bool S=0;
        bool H=0;
    string name;
    if(argc<2){
                printf("########## error #########\n");
        printf("-n : Origin pagerank\n") ;
        printf("-s : Sparse Matrix\n") ;
        printf("-i : Initial Value\n") ;
        printf("-e : Quadratic Extrapolation\n") ;
        printf("-f : Fix Converged nodes\n") ;
                printf("########## error #########\n");
        return 0 ;
        }
        for(argc--,argv++;argc>0;argc-=1,argv+=1){
        if(!strcmp(*argv,"-n")){
            name+="naive";
                        N=1;
                }
        else if(!strcmp(*argv,"-s")){
            name+="sparse";
                        S=1;
        }
        if(!strcmp(*argv,"-i")){
            name+="_init";
                        I=1;
        }
        if(!strcmp(*argv,"-e")){
            name+="_extra";
                        E=1;
        }
        if(!strcmp(*argv,"-f")){
            name+="_is_Converge";
                        F=1;
        }
    }
        QueryPerformanceFrequency(&frequ);
        create_idx();
        build_statistics(W);
        QueryPerformanceCounter(&start_time);
        Init_value(I);
        if(N)PageRank_origin();
        if(S)PageRank_sparse(E,F);
        QueryPerformanceCounter(&end_time);
    cout<<(double)(end_time.QuadPart-start_time.QuadPart)/frequ.QuadPart
       <<endl;
        output_to_txt(name+".txt");
        return 0;
}
```

Header file for the accelerated pagerank algorithm, defining some variables and functions.
accelerated_pagerank.h:

```
//pagerank
#include <iostream>
#include <map>
#include <algorithm>
#include <vector>
#include <windows.h>
#include <string>
#include <fstream>
#include <cstring>
using namespace std;
#define max_iteration 100000
#define epslion 1e-10
#define epslionIndex 10
#define maxnodes 8000
#define maxedges 403689
#define maxindegree 10000
```

```
#define Poblity 0.15
#define maxlength 20
LARGE_INTEGER frequ;
LARGE_INTEGER start_time, end_time;
map<int, string> id_2_name;//id to name
map<string, int> name_2_id;//name to id
typedef pair<int, double> weigh_edges;//edges have weights
vector<weigh_edges> wedges_vec;
unsigned short spares_matrix_cite[maxnodes][maxindegree];
float matrix[maxnodes][maxnodes];
int no_outlinks[maxnodes];
int num_total;
int num_of_edges;
int num_of_nooutlinks;
int count_outlinks[maxnodes];//cnt of outlinks of node
int count_inlinks[maxnodes];//cnt of inlinks of node
int count_outlinks_in[maxnodes];
bool is_Converge[maxnodes];
float weight[maxnodes][maxindegree];
double val[maxnodes];
double newval[maxnodes];
//The difference between the two iterations
double last_difference[maxnodes];
int dif[maxnodes];
double the_maxchange[max_iteration];//Maximum change in weights per
    iteration
//the number of the largest difference in each generation
int number[max_iteration];
//the ovalues of the point
double v[max_iteration];
double vn[max_iteration];
int iterate[epslionIndex+1];//number of iteration
double time[epslionIndex+1];//using time

//compare
struct cmp_fun {
        bool operator()(const weigh_edges& the_left, const weigh_edges&
            the_right) {
        return the_left.second > the_right.second;
        }
};
//read data
```

```
void create_idx();
void build_statistics(int weighted);
//init the value of matrix
void Init_value(int Init);
//check if converged
int check(int iter);
//Quadratic Extrapolation method
void Quadratic_Extrapolation();
//the origin pagerank alg
void PageRank_origin();
//pagerank alg using sparse matrix
void PageRank_sparse(int Extra, int Fix);
// Function to generate the result and save it to a file
void output_to_txt(string name);
```

accelerated_pagerank.cpp implements the functions in the header file.

accelerated_pagerank.cpp:

```
#include "accelerated_pagerank.h"

//read data
void create_idx(){
        FILE *file1=fopen("../Wiki-Vote-id.txt", "r");
        FILE *file2=fopen("../Wiki-Vote.txt", "r");
        if(file1==NULL||file2==NULL){
                cout<<"open input file error!!!!!!"<<endl ;
                return;
    }
        int i;
        for(i=1;i<=maxnodes;i++){
                char name[maxlength];
                if(fgets(name,20,file1)==0)break;
                name[strlen(name)-1]=0;
                id_2_name[i]=name;
                name_2_id[name]=i;
    }
    num_total=i-1 ;
        for(i=1;i<=maxedges;i++){
                char temp[2*maxlength+5];
                char name1[maxlength];
                char name2[maxlength];
                int j=0;
                if(fgets(temp,2*maxlength+5,file2)==0)break ;
                for(j=0;temp[j+1]!='\t';j++){
                        name1[j]=temp[j];
                }
                name1[j]=0;
                int j0=j+2;
                for(j=j0;temp[j-1]!=0;j++){
                        name2[j-j0]=temp[j];
                }
                name2[j-j0-2]=0;
                if(name_2_id.find(name1)==name_2_id.end())continue ;
                if(name_2_id.find(name2)==name_2_id.end())continue ;
                int index1=name_2_id[name1];
                int index2=name_2_id[name2];
                //Record this citation relationship
        spares_matrix_cite[index2][count_inlinks[index2]]=index1;
        count_inlinks[index2]++;
        count_outlinks[index1]++;
        //index1 link to index2
        matrix[index2][index1]+=1;
    }
    num_of_edges=i-1;
    fclose(file1);
    fclose(file2);
}

void build_statistics(int weighted)
{
        cout<<"########################################\n";
        cout<<"the number of pages: "<<num_total<<endl;
        cout<<"the number of links: "<<num_of_edges<<endl;
        cout<<"########################################\n";
        int the_max_num=0;
        //Initial state, each point has the same weight. All points have weights that sum to 1
        for(int i=1;i<=num_total;i++){
                if(count_inlinks[i]>the_max_num)the_max_num=count_inlinks[i];
        }
        for(int i=1;i<=num_total;i++){
                for(int j=0;j<count_inlinks[i];j++){
                        count_outlinks_in[spares_matrix_cite[i][j]]+=count_inlinks[i];
```

```
                    }
        }
        if(weighted==0){
                    for(int i=1;i<=num_total;i++){
                            for(int j=0;j<count_inlinks[i];j++){
                                    weight[i][j]=(double)1/count_outlinks[
                                        spares_matrix_cite[i][j]] ;
                            }
                    }
        }
        else{
                    for(int i=1;i<=num_total;i++){
                            for(int j=0;j<count_inlinks[i];j++)weight[i][j
                                ]=(double)count_inlinks[i]/count_outlinks_in
                                [spares_matrix_cite[i][j]] ;
                    }
        }
        //Calculate the transfer matrix
    for(int i=1;i<=num_total;i++){
        for(int j=1; j<=num_total;j++){
                            if(matrix[i][j]!=0){
                                    if(weighted==0)matrix[i][j]=matrix[i][j
                                        ]*(1-Poblity)/count_outlinks[j];
                                    else matrix[i][j]=matrix[i][j]*(1-
                                        Poblity)*count_inlinks[i]/
                                        count_outlinks_in[j];
                            }
                            matrix[i][j]+=(double)Poblity/num_total;
                    }
         }
    //Record points that have not outlinks
    for(int i=1; i<=num_total;i++){
        if(count_outlinks[i]==0){
            no_outlinks[num_of_nooutlinks]=i;
            num_of_nooutlinks++;
                            // Update the transfer matrix by connecting the
                                points that do not have an edge to all the
                                points.
            for(int j=1;j<=num_total;j++)matrix[j][i]+=(double)(1-
                Poblity)/num_total;
        }
    }
```

```
}

//init the value of matrix
void Init_value(int Init){
        if(Init==0){
                for(int i=1;i<=num_total;i++){
                        val[i]=1/double(num_total);
                }
        }
        if(Init==1){
                int inlinks_sum=0;
                for(int i=1;i<=num_total;i++){
                        inlinks_sum+=count_inlinks[i]+1 ;
                }
                for(int i=1;i<=num_total;i++){
                        val[i]=(count_inlinks[i]+1)/double(inlinks_sum);
                }
        }
}

//check if converged
int check(int iter){
        static float threshold=0.001;   //Threshold of convergence
        static int power=3;     //the power of Threshold
        the_maxchange[iter]=0;  //max change of pagerank
        //Calculate the maximum change
        for(int i=1;i<=num_total;i++){
                if(val[i]-newval[i]>the_maxchange[iter]){
                        number[iter]=i;
                        v[iter]=val[i];
                vn[iter]=newval[i];
                        the_maxchange[iter]=val[i]-newval[i];
                }
                else if(newval[i]-val[i]>the_maxchange[iter]){
                number[iter]=i;
                        v[iter]=val[i];
                        vn[iter]=newval[i];
                        the_maxchange[iter]=newval[i]-val[i];
                }
                //some node Converge
                if(newval[i]-val[i]<epslion/10&&-newval[i]+val[i]<
                    epslion/10)is_Converge[i]=1;
```

```
            val[i] = newval[i] ;
    }
            //start record time
            if(the_maxchange[iter]<threshold){
                    threshold/=10;
                    iterate[power]=iter;
                    QueryPerformanceCounter(&end_time);
            time[power++]=(double)(end_time.QuadPart-start_time.QuadPart)/
                freqU.QuadPart;
            }
    //Determine whether the convergence condition is satisfied
    if(the_maxchange[iter]<epslion)return 1;
    else return 0;
}

//Quadratic Extrapolation method
void Quadratic_Extrapolation(){
        double lamda;
        double sum1=0;
        double sum2=0;
        //Calculated as shown in the formula
        for(int i=1;i<=num_total;i++){
                sum1+=last_difference[i]*last_difference[i];
                sum2+=last_difference[i]*dif[i];
        }
        lamda=sum2/sum1;
        for(int i=1;i<=num_total;i++){
                newval[i]=(lamda*val[i]-newval[i])/(lamda-1);
        }
}

//the origin pagerank alg
void PageRank_origin(){
        //Calculated as shown in the formula
        for(int iter=1;iter<=max_iteration/4;iter++){
                for(int i=1;i<=num_total;i++){
                        newval[i]=0;
                        for(int j=1;j<=num_total;j++)newval[i]+=matrix[i
                            ][j]*val[j];
                }
                if(check(iter)==1)break;
    }
```

```
}

//pagerank alg using sparse matrix
void PageRank_sparse(int Extra, int Fix){
    for (int iter=1;iter<max_iteration;iter++){
        double sum_val0=0;
        // Calculate the sum of values for nodes with no outgoing links
        for (int j=0;j<num_of_nooutlinks;j++)sum_val0+=val[no_outlinks[j
            ]];
        // Calculate the new PageRank values for each node
        for (int i=1;i<=num_total;i++){
            if(Fix&&is_Converge[i])continue;
            // Nodes with outgoing links
            // If the node has outgoing links, assign it a value based
                on the sum of values for nodes with no outgoing links
            if(count_outlinks[i]!=0)newval[i]=Poblity/double(num_total)
                +(1-Poblity)*sum_val0/double(num_total-1);
            // If the node has no outgoing links, assign it a value
                based on the sum of values for nodes with no outgoing
                links
            else newval[i]=Poblity/double(num_total)+(1-Poblity)*(
                sum_val0-val[i])/double(num_total-1);
            // The second part of the PageRank value is obtained by "
                absorbing" from all incoming edges to the node
            for(int j=0;j<count_inlinks[i];j++){
                // The new PageRank value of node i is updated by adding
                     the probability of not performing a random walk,
                // the weight of the j-th incoming edge of node i, and
                    the PageRank value of the incoming node
                newval[i]+=(1-Poblity)*val[spares_matrix_cite[i][j]]*
                    weight[i][j];
            }
        }
        // Use quadratic extrapolation to accelerate the convergence (if
             enabled)
        if(Extra==1){
            if(iter==1){
                for(int i=1;i<=num_total;i++)
                    dif[i]=newval[i]-val[i];
            }
                        else{
                for(int i=1;i<=num_total;i++){
```

```cpp
                    last_difference[i]=dif[i];
                    dif[i]=newval[i]-val[i];
                }
                Quadratic_Extrapolation();
            }
        }
        // Check if the PageRank values have converged
        if(check(iter)==1)break;
    }
}

// Function to generate the result and save it to a file
void output_to_txt(string name) {
    // Calculate the sum of all node values. If computed correctly, the sum of all node values should be 1.
    double allsum=0;
    for(int i=1;i<=num_total;i++) {
        // Insert the node and its value into a vector for sorting
        wedges_vec.push_back(make_pair(i,val[i]));
        allsum+=val[i];
    }
    // Sort the nodes based on their values
    sort(wedges_vec.begin(),wedges_vec.end(),cmp_fun());
    // Open the result file
    cout << "############ saving result ################" << endl;
    FILE *file3;
    FILE *file4;
    file3=fopen((string("output/PageRank_with_")+name).c_str(),"w");
    // Convert the node IDs back to author names and output the names and PageRank values
    for(int i=0;i<num_total;i++)fprintf(file3,"%s\t%f\n",id_2_name[wedges_vec[i].first].c_str(),wedges_vec[i].second);
    fclose(file3);
    // Output the convergence information for each iteration
    file4=fopen((string("output/statistics_")+name).c_str(),"w");
    fprintf(file4, "acc\ttimes\n");
    for(int power=3;power<=epslionIndex;power++)fprintf(file4,"%d\t%llf\n",power,time[power]);
    fprintf(file4,"acc\titeration\n");
    for(int power=3;power<=epslionIndex;power++)fprintf(file4,"%d\t%d\n",power,iterate[power]);
    fprintf(file4,"iteration\tmax change node\tlast val\tnew val\tchange
        \n");
    for(int i=1;i<=max_iteration;i++){
        fprintf(file4,"%d\t%d\t%.30llf\t%.30llf\t%.30llf\n",i,number[i],
            v[i],vn[i],the_maxchange[i]);
        if(the_maxchange[i]<epslion)break;
    }
    fclose(file4);
    wedges_vec.clear();
}
```