



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

大数据计算及应用实验报告

PageRank 实验报告

刘哲泽、叶潇晗、李翔

年级：2021 级

指导教师：杨征路

2024 年 4 月 29 日

目录

一、 简介	1
(一) PageRank 算法	1
(二) 数据集	1
(三) 作业完成情况	1
二、 基础 Page Rank 算法实现	1
(一) 小节内容	1
(二) 概念介绍	2
(三) 优化稀疏矩阵	4
(四) 考虑死结点和蜘蛛网陷阱节点	5
(五) 基础 PageRank 算法的代码实现	7
三、 分块 Page Rank 算法实现	9
(一) 小节内容	9
(二) 算法介绍	9
(三) 分块 PageRank 算法的代码实现	10
四、 测试结果及部分实验结果	13
(一) 基础 PageRank 算法	13
(二) 分块 PageRank	14
(三) 不同方法结果分析	14
五、 总结	16

一、简介

(一) PageRank 算法

在现代数据科学中，许多数据结构都可以表示为图，如互联网、社交网络等。这些图结构中的数据为机器学习提供了丰富的理论和应用场景。其中，PageRank 算法是图链接分析的经典代表，它是图数据上的无监督学习方法的典范。PageRank 算法基于图论和概率统计的原理，通过分析网页之间的链接关系来确定网页的权重和排名。

PageRank 算法的核心思想是：一个网页的重要性取决于其他网页对它的引用和链接情况，即一个网页被越多其他网页链接，那么它的重要性就越高。同时，一个链接来自重要性高的网页的话，那么这个链接的价值也会更高。

PageRank 算法通过迭代计算的方式来确定每个网页的权重值，最终得到一个稳定的排名结果。在计算过程中，每个网页的初始权重值可以设定为相等，然后根据链接关系进行迭代更新，直到收敛为止。

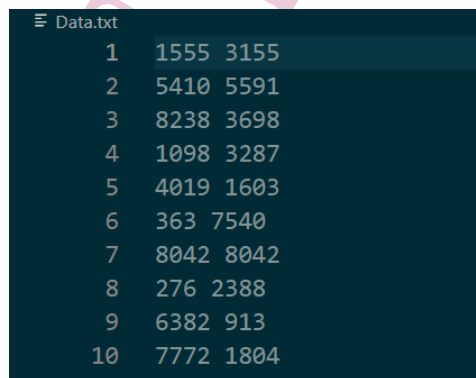
PageRank 算法在谷歌搜索引擎中起到了重要的作用，帮助用户找到更相关和权威的网页。同时，该算法也被广泛应用于其他领域，如社交网络分析、推荐系统等。

(二) 数据集

使用的数据集为 *Data.txt*，根据该数据集生成的图网络，有 8297 个节点和 135737 条边，该数据集文件的数据存储格式为：

FromNodeID ToNodeID

Data.txt 中每行表示一条链接边，从 *FromNodeID* 指向 *ToNodeID*。*Data.txt* 部分内容如图1所示。



	FromNodeID	ToNodeID
1	1555	3155
2	5410	5591
3	8238	3698
4	1098	3287
5	4019	1603
6	363	7540
7	8042	8042
8	276	2388
9	6382	913
10	7772	1804

图 1: 数据集

(三) 作业完成情况

我们小组完成的内容如表1所示。

二、基础 Page Rank 算法实现

(一) 小节内容

该小节介绍的内容如表2所示。

序号	作业要求	完成情况
1	基础 Page Rank 算法	✓
2	考虑 dead ends 和 spider trap 节点	✓
3	优化稀疏矩阵	✓
4	实现分块计算	✓

表 1: 作业完成情况

序号	作业要求	完成情况
1	基础 Page Rank 算法	✓
2	考虑 dead ends 和 spider trap 节点	✓
3	优化稀疏矩阵	✓

表 2: 基础 Page Rank 算法内容

(二) 概念介绍

PageRank 算法解决的是有向图网络中的问题，如图2所示，我们介绍几个有向图网络中的概念。

- **入度**：指向该节点的链接数。
- **出度**：由该节点指出的链接数。

例如，在图2中，a 节点的入度为 2，a 节点的出度为 2。我们将节点 i 的出度用 d_i 表示，因此节点 i 的权重即为 $\frac{1}{d_i}$ 。

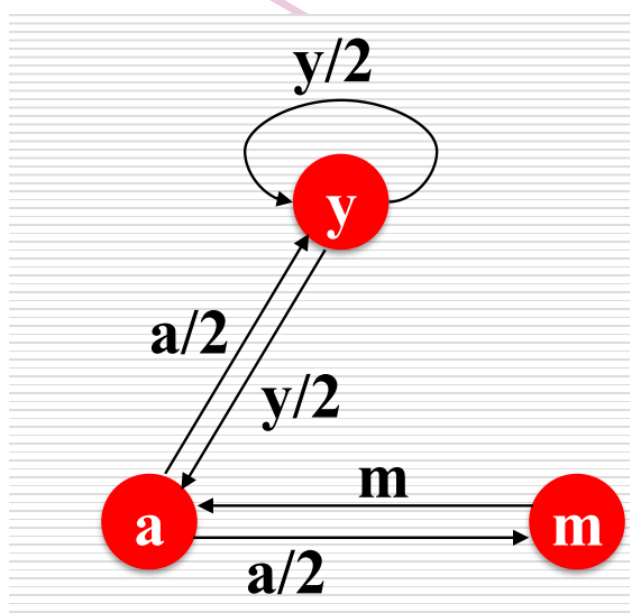


图 2: 有向图

记节点 i 的排名分数为 r_i ，因此我们可以定义网页 j 的排名分数计算公式为：

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i} \quad (1)$$

因此，图2可以表示为三个等式：

$$\begin{cases} r_y = \frac{r_y}{2} + \frac{r_a}{2} \\ r_a = \frac{r_y}{2} + r_m \\ r_m = \frac{r_a}{2} \end{cases} \quad (2)$$

此方程组含有三个方程，共有三个未知数，因此没有唯一解，我们添加一个约束：

$$\begin{cases} r_a + r_y + r_m = 1 \end{cases} \quad (3)$$

解此方程，就可以得到图2中三个节点的得分。但是不希望解方程组，而是将其表示成为可迭代的形式。我们将其表示为形如 $M * r = r$ 的矩阵形式：

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix} \quad (4)$$

经过迭代，可以求出各个节点最终的 PageRank 得分，迭代过程如下：

假设有 N 个网页

初始化： $r^{(0)} = [1/N, \dots, 1/N]^T$

迭代： $r^{(t+1)} = M * r^{(t)}$

当 $|r^{(t+1)} - r^{(t)}|_1 < \epsilon$ 时停止迭代

可以证明，一般情况下，网页的得分可以迭代至收敛。但是有一些特殊情况会导致迭代出现异常，比如下文将要介绍的死结点和蜘蛛网陷阱节点。

下面给出我们实现的 PageRank 类包含的属性和方法，定义在 *PageRank.h* 头文件中：

PageRank.h

```
1 typedef long double long_d;
2
3 class PageRank
4 {
5 private:
6     /*
7     beta为PageRank算法中的 teleport parameter
8     */
9     double beta;
10
11     /*
12     */
13     double epsilon;
14
15     /*
16     分块数量
17     */
```

```

18  int block_num;
19
20  /*
21  graph 存储图网络 FromNodeID -> ToNodeID
22  page_rank 存储每个节点的PageRank值
23  page_rank_sorted 存储根据value排序后的数据
24  */
25  unordered_map<int, unordered_set<int>> graph;
26  unordered_map<int, long_d> page_rank;
27  vector<pair<int, long_d>> page_rank_sorted;
28
29  /*
30  输入文件和输出文件
31  */
32  char* input_file;
33  char* output_file;
34  char* output_file_all;
35
36  //节点数量
37  int node_count=0;
38
39  public:
40      PageRank(double beta, double epsilon, int block_num, char *input_file,
41              char *output_file, char* output_file_all);
42      ~PageRank();
43
44      void load_data(); //加载 input_file 文件
45      void rank_base(); // 基本的 PageRank 算法 (不含 优化稀疏矩阵 和 实现分块计
46                          算)
47      void load_data_2();
48      void rank_block();
49      long_d compute_L1(unordered_map<int, long_d> old_rank, unordered_map<int,
50                          long_d> new_rank); // 计算 L1 误差值
51      void mysort(); // 排序
52      void write_to_result(); // 将排序结果写入 output_file 和 output_file_all
53  };

```

(三) 优化稀疏矩阵

由于有很多节点之间没有链接，因此矩阵 M 中有大量的空间被浪费，我们对这种情况下的稀疏矩阵进行优化。我们将稀疏矩阵存储为源节点及其目的节点集合的形式。我们在代码中，用如下的数据结构存储稀疏矩阵 M ：

稀疏矩阵表示

```

1  unordered_map<int, unordered_set<int>> graph;

```

我们使用 `load_data()` 函数，来进行基础 PageRank 算法的数据读取，即将 `Data.txt` 中的数据读取到 `graph` 中，定义在 `PageRank.cpp` 文件中：

```
load_data()

1 void PageRank::load_data()
2 {
3     /*
4     加载文件中的数据
5     */
6     ifstream inf;
7
8     inf.open(this->input_file, ios::in);
9
10    int FromNodeID, ToNodeID;
11    // 使用文件读取操作的成功与否作为循环的终止条件
12    while (inf >> FromNodeID >> ToNodeID) // Data.txt 共有 135737 行
13    {
14        /*
15        Data.txt
16        一共 135737 行;
17        最小节点为 1;
18        最大节点为 8297;
19        */
20        this->graph[FromNodeID].insert(ToNodeID);
21        this->graph[ToNodeID]; // 若为空则会初始化; 若不为空没有影响
22    }
23    this->node_count = this->graph.size();
24    inf.close();
25 }
```

在迭代时，需要考虑新的迭代计算方式，我们将在接下来的小节中介绍。

(四) 考虑死结点和蜘蛛网陷阱节点

死结点 (dead end) 是指在网络图中，出度为 0 的节点。蜘蛛网陷阱 (spider traps) 是指在网络图中，所有的外向链接都在同一个节点组中。如图3所示。

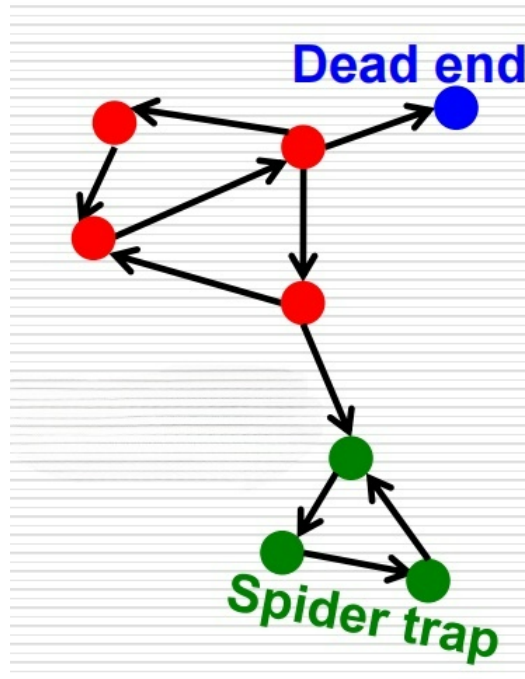


图 3: 死结点和蜘蛛网陷阱

死结点会导致整个网络图各个节点的得分，在多次迭代之后消失。而蜘蛛网陷阱，则会聚集所有的得分，导致得分汇聚在节点组中。这两种情况，都会严重影响最后的结果。我们使用随机跳转（teleport）来解决这个问题。随机跳转就是赋予每个节点 β 的概率，到达原本没有链接边的节点（如图4和图5所示）。

死结点的随机跳转：如图4所示， m 是一个死结点，我们赋予死结点 m 概率 β 跳转到其他节点，也就是说， m 的出度原本为 0，而我们让他有 β 的概率有出度 3。

蜘蛛网陷阱结点的随机跳转：如图5所示， m 是一个蜘蛛网陷阱结点，我们让其有 β 的概率有链接到节点 a 和 y 的边。

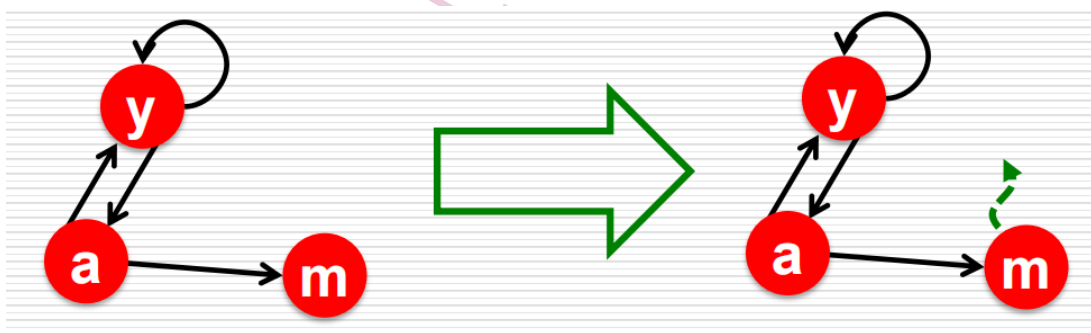


图 4: 死结点的随机跳转

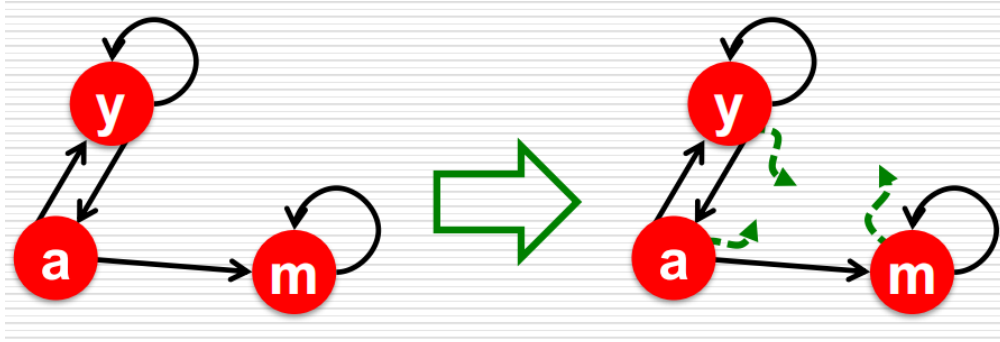


图 5: 蜘蛛网陷阱的随机跳转

将上述过程表示为一个公式如下所示:

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N} \quad (5)$$

根据上述公式, 我们实现了如图6所示的算法:

- **Set:** $r_j^{old} = \frac{1}{N}$
- **repeat until convergence:** $\sum_j |r_j^{new} - r_j^{old}| > \varepsilon$
 - $\forall j: r_j^{new} = \sum_{i \rightarrow j} \beta \frac{r_i^{old}}{d_i}$
 $r_j^{new} = 0$ if in-degree of j is 0
 - **Now re-insert the leaked PageRank:**
 $\forall j: r_j^{new} = r_j^{new} + \frac{1-S}{N}$ **where:** $S = \sum_j r_j^{new}$
 - $r^{old} = r^{new}$

图 6: 基础 PageRank 算法

(五) 基础 PageRank 算法的代码实现

首先, 我们定义了一个 `compute_L1()` 函数, 来计算新得分和旧得分的 L1 距离:

```

compute_L1()
1 long_d PageRank::compute_L1(unordered_map<int, long_d> old_rank,
  unordered_map<int, long_d> new_rank){
2     // 计算L1值
3     long_d L1=0.0;
4     for(int i=1;i<=this->node_count;i++){
5         L1 += abs(old_rank[i]-new_rank[i]);
6     }
7     return L1;

```

8 }

然后,我们根据图6所示的算法,实现了 `rank_base()` 函数。首先,初始化 `this->page_rank` 作为 r^{old} , 同时定义一个 r^{new} ; 接着进入一个 `while` 循环, 直到 $L1 = compute_L1(this->page_rank, new_rank)$ 小于特定的 `epsilon` 时, 结束循环; 在 `while` 循环中, 首先将 r^{new} 初始化为 0, 然后按照图6的第一行算法所示, 计算随机跳转的得分; 然后再将结果加上 $\frac{1-S}{N}$ 即可。这个算法可以解决死结点和蜘蛛网陷阱结点问题。

`rank_base()`

```

1 void PageRank::rank_base(){
2     // r^(old)
3     for(int i=1;i<=this->node_count;i++){
4         //初始化节点的PageRank值
5         this->page_rank[i]=1.0/this->node_count;
6     }
7     // r^(new)
8     unordered_map<int, long_d> new_rank=this->page_rank;
9
10    long_d L1 = 1.0; //计算 r^(old) 和 r^(new) 的L1差值
11    long_d S = 0.0; //计算 r^(new) 的和
12
13    int round=1; //记录迭代轮数
14
15    // 重复计算直到收敛 (差值和小于epsilon)
16    while(L1 > this->epsilon){
17        // 初始化
18        for(auto [key, value]: new_rank){
19            new_rank[key] = 0.0;
20        }
21
22        // 解决 spider traps
23        for(int j=1;j<=this->node_count;j++){
24            // 计算 r^(new)_j
25            unordered_set<int> from_j = graph[j]; // 节点 j 的所有
                出度节点
26            int out_degree = from_j.size();
27
28            for(unordered_set<int>::iterator it=from_j.begin(); it
                !=from_j.end(); it++){
29                // 更新 *it 节点
30                new_rank[*it] += this->beta*(this->page_rank[
                    j]*1.0/from_j.size());
31            }
32        }
33        // 解决 dead ends
34        // 计算 new_rank的和
35        S = 0.0;
36        for(auto [key, value]: new_rank){

```

```

37         S += value;
38     }
39     // 更新
40     for (int j=1;j<=this->node_count;j++){
41         new_rank[j] = new_rank[j] + (1.0 - S)/(1.0 * this->
42             node_count);
43     }
44     L1 = compute_L1(this->page_rank, new_rank);
45     this->page_rank = new_rank;
46     // cout<<" " <<S<<endl;
47     cout<<" 已经迭代了 " <<round++<<" 轮... " <<" 现在的 L1 距离为 " <<
48         L1<<endl;
49 }

```

三、 分块 Page Rank 算法实现

(一) 小节内容

该小节介绍的内容如表3所示。

序号	作业要求	完成情况
4	实现分块计算	✓

表 3: 小节内容

(二) 算法介绍

类似于 databases 中的嵌套循环链接，分块算法将 r_{new} 分成适合存于内存的 k 块，然后对于每个块扫描 M 和 r_{old} ，每个条带包括目的节点在相应的 r_{new} 的块中的节点。如下图分块：

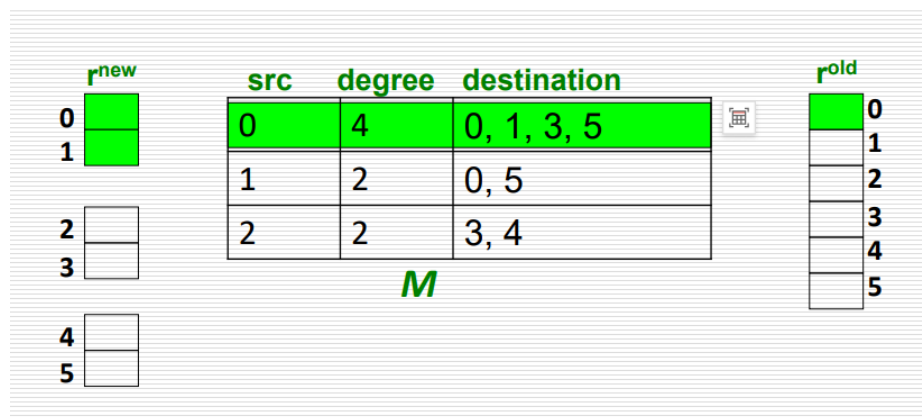


图 7: 分块前数据集

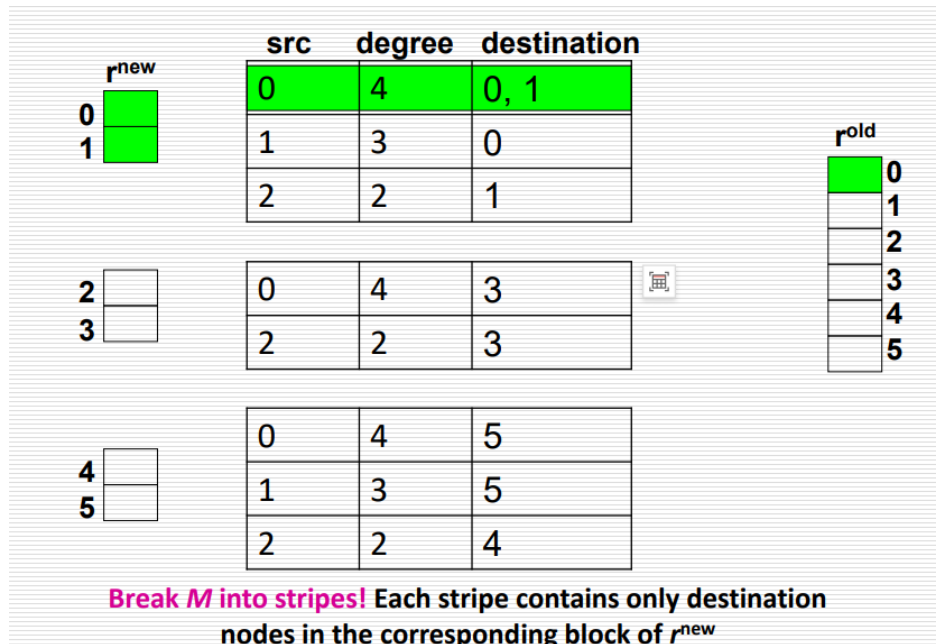


图 8: 分块后数据集

在计算时，对每个子图执行局部 PageRank 计算，对每个子图完成计算后实现对整个图的 PageRank 计算。使用分块算法通常有如下一些优点：

1. 降低计算复杂度：将大问题分解成小问题，每个子图的计算复杂度较低，更容易处理。
2. 节省内存开销：每个子图的规模较小，需要的内存资源更少，能够在内存有限的情况下运行。
3. 适用于大规模图：在处理大规模图时，分块的 Pagerank 算法能够更有效地应对内存和计算资源的限制，使得算法能够扩展到更大的网络规模。

(三) 分块 PageRank 算法的代码实现

分块算法主要要对数据集进行处理，在读取数据时根据不同块将其分开。首先根据块数和节点数计算出每块的数量，在这里将每块命名为 `file_num`。每块对应输出一个文件，文件中存储分块后该块的数据信息，命名为 `block_num`。对于分块算法，我们在处理数据时即将其 rank 值初始化为 $1/\text{max_node_num}$ 。

```

load_data_2()

1 void PageRank::load_data_2() {
2     int max_node_num=0;
3     int block_num=this->block_num;
4     /*加载文件中的数据*/
5     ifstream inf;
6     inf.open(this->input_file, ios::in);
7     int FromNodeID, ToNodeID;
8     //cout<<1;
9     while (inf >> FromNodeID >> ToNodeID) //Data.txt 共有 135737 行{
10    max_node_num=std::max(std::max(FromNodeID, ToNodeID), max_node_num);
11    }
12    inf.close();

```

```

13 // 每块的数量
14 int step = max_node_num/block_num;
15 // 每块的名字
16 auto belong_file = [&](int node_num)
17 {
18     int file_num=node_num/step+1;
19     file_num = std::min(file_num,block_num);
20     return file_num;
21 };
22 std::ofstream out;
23 // 每块输出一个文件, 文件中对该块存一个表, 每行为node:to_node1,to_node2
24     .....
25 for(int i=1;i<=block_num;i++){
26     this->graph.clear();
27     inf.open(input_file);
28     while(inf>>FromNodeID>>ToNodeID){
29         if(belong_file(FromNodeID)==i)
30         {
31             this->graph[FromNodeID].insert(ToNodeID);
32         }
33         if(belong_file(ToNodeID)==i)
34         {
35             this->graph[ToNodeID];
36         }
37     }
38     this->node_count+=this->graph.size();
39     out.open("block"+std::to_string(i));
40     for(auto &edge:this->graph)
41     {
42         out<<edge.first;
43         for(auto &ToNodeID:edge.second)
44         {
45             out<<" ";<<ToNodeID;
46         }
47         out<<"\n";
48     }
49     inf.close();
50     out.close();
51     for(auto &[FromNodeID,ToNodeID]:this->graph)
52     {
53         this->page_rank[FromNodeID]=1.0/max_node_num;
54         //node2mutex[FromNodeID];
55     }
56 }
57 //cout<<1;
58 }

```

然后还需改变 PageRank 算法, 和基础的 PageRank 算法的区别在于要嵌入一重循环, 对子图挨个处理, 处理的方式和基础 PageRank 算法基本相同。需要注意的是, 在对所有子图处理结束之后, 需要处理图中那些未连接的节点 (死节点), 在这里使用远程传播的方式即可。

rank_block()

```

1 void PageRank::rank_block() {
2     // r^(old) 在 load_data 里赋值
3     // r^(new)
4     unordered_map<int, double> new_rank=this->page_rank;
5     double L1 = 1.0; // 计算 r^(old) 和 r^(new) 的 L1 差值
6
7     int round=1; // 记录迭代轮数
8     // cout<<1;
9
10    while(L1>this->epsilon)
11    {
12        for(auto &[key, value]: page_rank){
13            new_rank[key]=0;
14        }
15        int from_node, to_node;
16        ifstream input;
17        string line;
18        int num=0;
19        for(int file_name=1; file_name<=block_num; file_name++)
20        {
21            this->graph.clear();
22            input.open("block"+std::to_string(file_name));
23            while(getline(input, line)){
24                std::istringstream ss(line);
25                ss>>from_node;
26                this->graph[from_node];
27                while(ss>>to_node)
28                {
29                    this->graph[from_node].insert(to_node);
30                }
31            }
32
33            input.close();
34
35            // r^new
36            for(auto &[from_node, to_nodes]: this->graph)
37            {
38                int out_degree=to_nodes.size();
39                for(auto to_node: to_nodes){
40                    new_rank[to_node]+=beta*(page_rank[from_node]/out_degree);
41                }

```

```
42         }
43     }
44
45     rank_type sum=0,inc=0;
46     for(auto &[node,rank]:new_rank)
47     {
48         sum+=rank;
49     }
50
51     inc=(1-sum)/node_count;
52
53     for(auto &[node,rank]:new_rank)
54     {
55         rank+=inc;
56     }
57
58     L1=compute_L1(this->page_rank,new_rank);
59     this->page_rank=new_rank;
60     //cout
61     std::cout<<"已经迭代了"<<round++<<"轮..."<<"现在的L1距离为"
        <<L1<<endl;
62 }
63 }
```

四、 测试结果及部分实验结果

(一) 基础 PageRank 算法

我们设置随机出链打开网页的概率为 0.85，收敛阈值 EPSILON 为 0.0001；误差采用绝对误差，即前后 pagerank 值差的绝对值表示。最大迭代次数为 100。运行实验结果前十名如下：

结点排序	结点序号	结点权重
1	2730	0.000871801
2	7102	0.000854476
3	1010	0.000849558
4	368	0.000835846
5	1907	0.000830538
6	7453	0.000820592
7	4583	0.000817828
8	7420	0.000810281
9	1847	0.000809945
10	5369	0.000805946

表 4: 基础算法测试结果

(二) 分块 PageRank

然后测试分块部分内容。设置 Random Teleports 值为 0.85, 收敛阈值 EPSILON 为 0.0001; 最大迭代次数为 100。r_new 被分成若干块, 每块大小为 compute_size=50, 稀疏矩阵因此相应地被分成 125 个 stipe。误差采用绝对误差, 即前后 pagerank 值差的绝对值表示; 实验结果前十名如下:

结点排序	结点序号	结点权重
1	2730	0.000871801
2	7102	0.000854476
3	1010	0.000849558
4	368	0.000835846
5	1907	0.000830538
6	7453	0.000820592
7	4583	0.000817828
8	7420	0.000810281
9	1847	0.000809945
10	5369	0.000805946

表 5: 分块算法测试结果

(三) 不同方法结果分析

TELEPORT=0.85 时, 将输出结果与分块结果进行比对我们发现: 基础 PageRank 算法得到的排名结果前 100 名与 block 分块得到的排名完全相同。证明了两种方法都是有效的

为探究随机出链概率 beta 对算法结果 (结点的 pagerank 计算结果) 的影响, 我们在平凡 PageRank 算法下记录不同 beta 下 TOP100 结点的分值。我们以 0.05 为步长, 设置了 0.6-0.9 八个数据的测试, 计算不同情况下的分数。在不同情况下也应该根据 beta 大小调整程序收敛次数, 保证程序结束时已经收敛。

下面选取 top100 分数展示如下

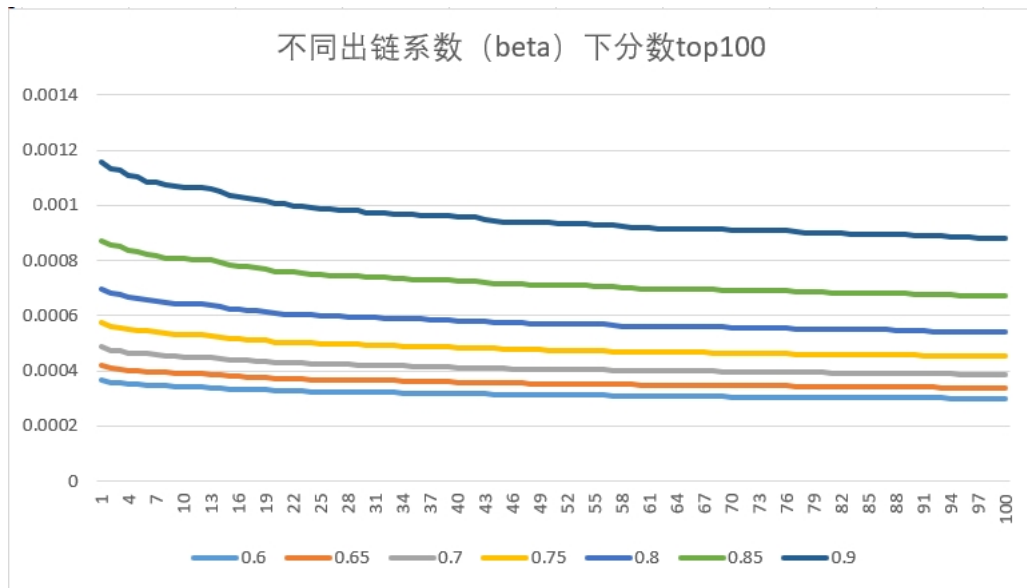


图 9: TOP100 结点分值

可以看到出链系数越大，TOP100 结点的分值普遍更大。从数学理论上分析结果与之相同，即 β 越大，其他网页分出去的可能性越高，分给结点对应网页的 PageRank 值也更大。实验结果与理论预期一致。

对于分块算法，为直观验证算法运行时间会随着块数的增加线性增长，我们在其他条件完全一致 ($\beta=0.85$ 且收敛阈值均为 10^{-5}) 只改变其分块数目，对分块 1 至 10 进行测试，如下表所示

块数	时间/s
1	2.7895
2	8.6509
3	10.9144
4	15.2788
5	16.6852
6	18.0065
7	19.9896
8	20.9922
9	21.6623
10	21.8985

表 6: 分块算法改变块数测试结果

为清晰展示随着块数增加运行时间变化的趋势，简要画折线图如下所示

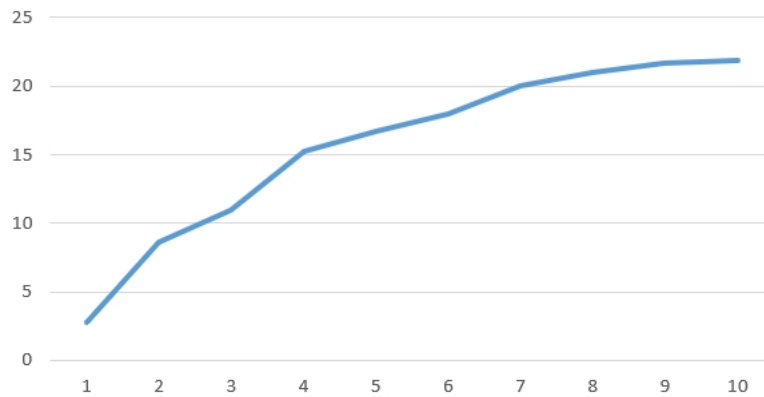


图 10: 运行时间与块数变化关系

从图中可知,运行时间随着分块数的增加而增加。每增加一块开始时间都有相同的增长,即有线性趋势,这是因为每增加一分块都增加一次内存 IO 读取时间,这一时间开始基本相同,到了后面由于分块数目增加,分块大小变小,读取时间减少,所以增长趋势逐渐减弱。因此,当迭代次数一定,PageRank 基本算法一定的情况下,总复杂度应当随着分块数增加而几近线性增加,块数变大后由于数据块包含数据减少,总时间可能收敛。

五、 总结

本实验我们对 Pagerank 算法这种基于图论的算法进行了基本学习,并考虑 dead ends 和 spider trap 节点两个网站运行中基本的问题,并进行稀疏矩阵优化、实现分块计算,最终在一定阈值内迭代至收敛。

在进行分块优化的计算中,也衍生出许多的问题。

1. 本质上来说,分块是进行内存优化,但并不会影响实验结果,但在同样的条件下将基础 PageRank 算法优化成分块以后 PageRank 值在小数点后九位相同,证明了这一结论。
2. 在改变 beta 的测试时,发现如果 beta 变化过大会影响结点排序这一结论,都有待结合实际案例进行更深入地探索。
3. 实际案例中仅有一个出链参数应该是远远不够的(上述改变 beta 就能影响结点排序就能看出这一点),应该考虑更多的出链跳转的情况,添加更多修正才能保证结点排序稳定。

总体来说,作为一个与查询无关的静态算法,PageRank 算法将所有网页的权重都可以通过离线计算获得;有效减少在线查询时的计算量,极大降低了查询响应时间,提高了 web 服务数据测算与挖掘的效率,但是由于忽略了主题相关性,并且给予旧网页初始更高的权重,导致最终排名在应用中可能存在问题,需要进一步改进。