



南開大學  
Nankai University

南 开 大 学

计算机与网络空间安全学院

大数据计算及应用

---

**Compute the PageRank scores**

---

学生：张龔、李潇、袁铭泽

学号：2012151、2012429、2012777

专业：计算机科学、密码班、信息安全

## 目录

一. 基本介绍 .....	3
1.什么是 pagerank? .....	3
2.基本思想 .....	3
3.算法原理 .....	3
二. 需解决的问题 .....	5
1.dead ends.....	5
2.spider trap.....	6
3.稀疏矩阵 .....	7
4.分块计算 .....	7
三. 数据集说明: .....	8
1. 基本形式: .....	8
2. 简单分析: .....	9
四. 关键代码讲解: .....	10
1. 稀疏矩阵优化以及存储结构优化: .....	10
2. dead ends 和 spider trap 节点处理: .....	12
3. 实现分块计算: .....	14
3.1. 分块方法: .....	14
3.2. 代码讲解: .....	15
3.3. 结果展示和比较: .....	18
4. 正确性验证: .....	19
4.1.与调用 network 库比较.....	19
4.2.与 $\beta$ 值分别设为 0.80 和 0.90 相比较 .....	20
五. 实验感想 .....	21

## 一. 基本介绍

### 1. 什么是 pagerank?

PageRank 算法最初作为互联网网页重要度的计算方法，1996 年由 Page 和 Brin 提出，并用于谷歌搜索引擎的网页排序。PageRank 有效地利用了 web 所拥有的庞大链接构造的特性。从网页 A 到网页 B 的链接可以看作是网页 A 对网页 B 的投票，Google 将根据这个投票数（即链接数）来判断页面的重要性。但 Google 不仅仅看投票，还会对投票的页面进行分析。“重要性”高的页面所投的票的价值应该更高，因为接受了这个投票会被理解为“重要的页面”。根据这样的分析，得到了高评价的重要页面也会被给予较高的 PageRank，即其网页等级会更高，在检索结果内的名词也会更高。所以，PageRank 是 Google 用于表示网页重要性的综合性指标。当然这项指标会与关键词相结合，重要性高的页面如果和检索词句中的关键词无关，那么该网页对于检索结果也就没有任何意义，所以 Google 使用了基于 PageRank 的文本匹配技术，使得能够检索出正确且重要的页面。

事实上，PageRank 可以定义在任意有向图上，后来被应用到社会影响力分析、文本摘要等多个问题。它的基本想法是在有向图上定义一个随机游走模型，即一阶马尔可夫链，描述随机游走者沿着有向图随机访问各个结点的行为。在一定条件下，极限情况访问每个结点的概率收敛到平稳分布，这时各个结点的平稳概率值就是其 PageRank 值，表示结点的重要度。PageRank 是递归定义的，PageRank 的计算可以通过迭代算法进行。

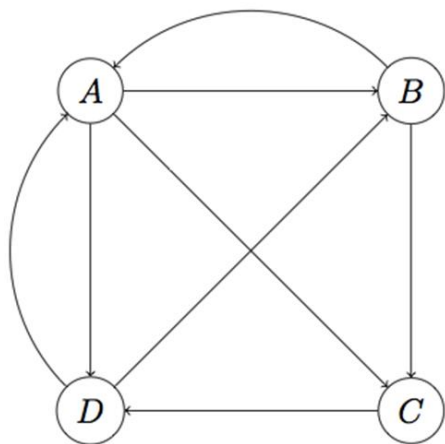
### 2. 基本思想

历史上，PageRank 算法作为计算互联网网页重要度的算法被提出。PageRank 是定义在网页集合上的一个函数，它对每个网页给出一个正实数，表示网页的重要程度，整体构成一个向量，PageRank 值越高，网页就越重要，在互联网搜索的排序中可能就被排在前面。

### 3. 算法原理

假设互联网是一个有向图，在其基础上定义随机游走模型，即一阶马尔可夫链，表示网页浏览者在互联网上随机浏览网页的过程。假设浏览者在每个网页依照连接出去的超链接以等概率跳转到下一个网页，并在网上持续不断进行这样的随机跳转，这个过程形成一阶马尔可夫链。PageRank 表示这个马尔可夫链的平稳分布。每个网页的 PageRank 值就是平稳概率。

现在假设世界上只有四张网页：A、B、C、D，简单情况下假设这个图是强连通的（从任一节点出发都可以到达另外任何一个节点）。则其抽象结构如下图所示：



然后需要用一种合适的数据结构表示页面间的链接关系。其实，PageRank 算法是基于这样一种背景思想：被用户访问越多的网页更可能质量越高，而用户在浏览网页时主要通过超链接进行页面跳转，因此我们需要通过分析超链接组成的拓扑结构来推算每个网页被访问频率的高低。最简单的，我们可以假设当一个用户停留在某页面时，跳转到页面上每个被链页面的概率是相同的。例如，上图中 A 页面链向 B、C、D，所以一个用户从 A 跳转到 B、C、D 的概率各为  $1/3$ ，且满足  $PR(A)=PR(B)+PR(C)+PR(D)$ ，即 A 的 PageRank 值是 B、C、D 的总和。设一共有 N 个网页，则可以组织这样一个 N 维矩阵：其中 i 行 j 列的值表示用户从页面 j 转到页面 i 的概率。这样一个矩阵叫做转移矩阵（Transition Matrix）。下面的转移矩阵 M 对应上图：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \\ 1/3 & 0 & 1 & 0 \end{bmatrix}$$

然后，设初始时每个页面的 rank 值为  $1/N$ ，在这个例子中就是  $1/4$ 。按 A-D 顺序将页面 rank 为向量 v：

$$v = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

注意，M 第一行分别是 A、B、C 和 D 转移到页面 A 的概率，而 v 的第一列分别是 A、B、C 和 D 当前的 rank，因此用 M 的第一行乘以 v 的第一列，所得结果就

是页面 A 最新 rank 的合理估计，同理，向量  $Mv$  的结果就分别代表 A、B、C、D 新 rank：

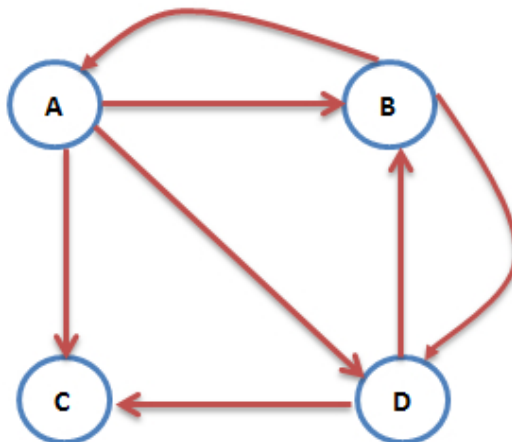
$$Mv = \begin{bmatrix} 1/4 \\ 5/24 \\ 5/24 \\ 1/3 \end{bmatrix}$$

然后用  $M$  再乘以这个新的 rank 向量，又会产生一个更新的 rank 向量。迭代这个过程，可以证明  $v$  最终会收敛，即  $v$  约等于  $Mv$ ，此时计算停止。最终的  $v$  就是各个页面的 PageRank 值。例如上面的向量经过几步迭代后，大约收敛在  $(1/4, 1/4, 1/5, 1/4)$ ，这就是 A、B、C、D 最后的 PageRank。

## 二. 需解决的问题

### 1.dead ends

上述  $v$  值满足收敛性源于图的强连通性。但实际上，web 并不是强连通的，甚至不是连通的，因为互联网中很多网页不指向任何网页。在这种情况下按上述过程继续迭代计算，会导致累计得到的转移概率最终被清零，即所有元素的 PageRank 几乎都为 0。这就是 PageRank 经典的 Dead Ends 问题。所谓 Dead Ends，就是不存在任何外链的节点，如下图：



其中 C 节点只有入度没有出度，即 C 节点不存在任何外链。这个有向图对应的初始转移矩阵为：

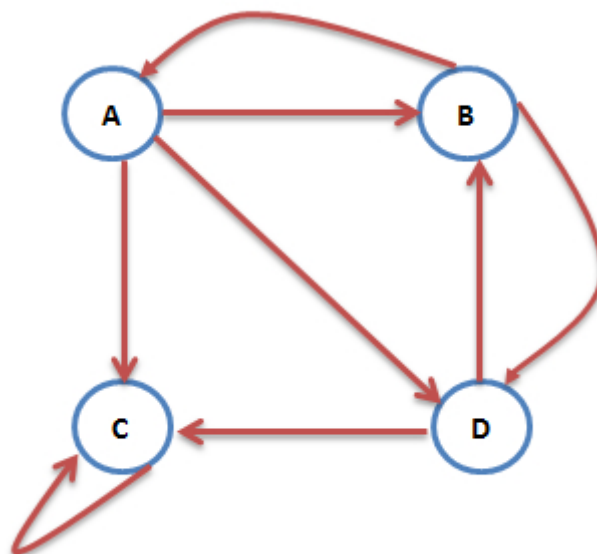
$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

按上述算法继续迭代，最终会导致所有元素均为 0：

$$v = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \Rightarrow \begin{bmatrix} 5/48 \\ 7/48 \\ 7/48 \\ 7/48 \end{bmatrix} \Rightarrow \begin{bmatrix} 21/288 \\ 31/288 \\ 31/288 \\ 31/288 \end{bmatrix} \dots \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

## 2.spider trap

Spider trap 陷阱问题指，有些网页不存在指向其他网页的链接，但存在指向自己的链接。如下图所示：



用户进入 C 网页后，就像跳进了陷阱，再也不能从 C 中出来，最终导致概率分布值全部转移到 C 上，使得其他网页的概率分布值为 0，从而整个网页排名失去意义。上图对应的转移矩阵为：

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

不断迭代后得到的结果为：

$$v = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \Rightarrow \begin{bmatrix} 3/24 \\ 5/24 \\ 11/24 \\ 5/24 \end{bmatrix} \Rightarrow \begin{bmatrix} 5/48 \\ 7/48 \\ 29/48 \\ 7/48 \end{bmatrix} \Rightarrow \begin{bmatrix} 21/288 \\ 31/288 \\ 205/288 \\ 31/288 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

### 3.稀疏矩阵

如果把真实网页组织成转移矩阵，由于互联网中网页数量庞大，那么矩阵将即为稀疏。极度稀疏的转移矩阵迭代相乘可能会使向量  $v$  非常不光滑，即一些节点拥有很大的 **rank**，而大多数节点 **rank** 值接近 0。同时，稀疏矩阵的特性会使得我们可以在存储和计算时通过巧妙的设计减少存储的空间以及计算的复杂度。

### 4.分块计算

分块计算的必要性在于：

- ① 原算法的时间开销大，每次迭代计算的时间开销为  $n^2$ ；
- ② 互联网中的数据大部分是分布式的，计算过程需要多次传递数据，造成网络负担太大；
- ③ 对于  $n$  维的稀疏矩阵，无论是计算还是存储都很浪费资源。所以考虑先算出局部 PageRank 的值。

分块式的 PageRank 算法大致思路为：

- ① 将大的网络有向图分数据块，计算每个子网络图的 Local PageRank；
- ② 根据各数据块之间的相关性计算由子网络图组成的缩略图的 BlockRank；
- ③ 将得到的 Local PageRank 和 BlockRank 按照一定能够的计算原则进行计算，得到一个新的  $n$  维 PageRank 矩阵，该矩阵一定比所有网页构成的稀疏矩阵缩减了很多倍；
- ④ 将新的  $n$  维 PageRank 多次迭代，得到最后收敛的 PageRank 向量。基于 MapReduce 框架实现的 PageRank 是常见的一种分块式计算方法。

针对上述问题，我们将在代码中一一解决，并在关键代码讲解部分中详细阐述各问题的解决方法。

### 三. 数据集说明:

#### 1. 基本形式:

第一次大作业: **Compute the PageRank scores on the given dataset**

Dataset: Data.txt

The format of the lines in the file is as follow:

**FromNodeID ToNodeID**

In this project, you need to report the **Top 100 NodeID** with their PageRank scores. You can choose different parameters, such as the teleport parameter, to compare different results.

One result you must report is that when setting the teleport parameter to **0.85**.

In addition to the basic **PageRank algorithm**, you need to implement the **Block-Stripe Update algorithm**.

Deadline: 2023.4.30

由 pdf 第一页前三行可知, 数据的形式为每一行表示有向图两个节点之间的指向关系, 使用节点的索引 ID 表示, 左边表出链节点, 右边表示入链节点, 形式如下:

```
1086 579
1549 5524
877 313
4564 4191
1350 1151
41 56
1922 3117
29 878
2485 2657
1166 4712
10 34
3092 2369
5016 5509
6229 8212
```



之后我们对于数据进行一个简单的观察：

## 2. 简单分析：

```
n=data_solve("Data.txt","data")  
m=data_decode("data")
```

```
max(max(n),max(m))
```

8297

```
len(n)
```

5496

```
len(m)
```

2037

我们先忽略以上的两个数据处理的函数（在关键代码讲解部分会进行说明），先说明 **n** 为{出链的节点：出度}字典，**m** 为字典（入链节点:[指向该节点的节点索引列表]），我们可以由 **max** 函数得到存在链的索引值最大（8297）的节点，这在后续的存储优化上提供了某些思路，以及得到存在出链的节点的数量为 5496，存在入链节点的数量为 2037。

由以上信息我们可得知节点状态可划分为 2 种，即孤立节点（不存在任何链），其他节点（存在出链或是入链）。

我们这里进行排名的节点不包括孤立节点。

## 四. 关键代码讲解:

### 1. 稀疏矩阵优化以及存储结构优化:

```
def data_solve(file, data):  
    m = defaultdict(list)  
    n = defaultdict(int)  
    #得到节点出度, 以及知道目的节点由哪些节点指向  
    with open(file, 'r') as f:  
        for i in f:  
            a,b = [int(x) for x in i.split()]  
            n[a] += 1  
            m[b].append(a)  
    #使用二进制读写来加速读取速度  
    with open(data, 'wb') as f:  
        for a,b in sorted(m.items()):  
            x=a.to_bytes(2, "little")  
            h=len(b)  
            x=x+h.to_bytes(2, "little")  
            for i in b:  
                x=x+i.to_bytes(2, "little")  
            f.write(x)  
    return n
```

这里为基本 pagerank 算法的载入数据部分的实现, 我们首先对于稀疏矩阵优化部分进行讲解, 对于转移概率矩阵我们进行优化的方式是使用二元组来表示稀疏矩阵即 (节点,[指向该节点的索引列表]), 这样在一定程度上减少了稀疏矩阵在内存的占用空间。

同时为了同后续的分块算法进行时间对比, 这里写了将元组表示的稀疏矩阵存入磁盘的操作。考虑到分块算法需要多次从磁盘中读写文件, 其中的时间开销不可小觑。通过网上查阅相关资料发现二进制文件的读写速度是要远大于文本文件, 所以我们采用了将元组进行进一步的处理并写入二进制文件的方式来实现类似压缩的效果。

在数据集说明中我们已经发现节点索引最大值为 8297, 即其表示范围是在 1 到 2 个字节之间, 所以我们这里的规则为使用固定的两个字节, 最终二进制文件读取后如下:

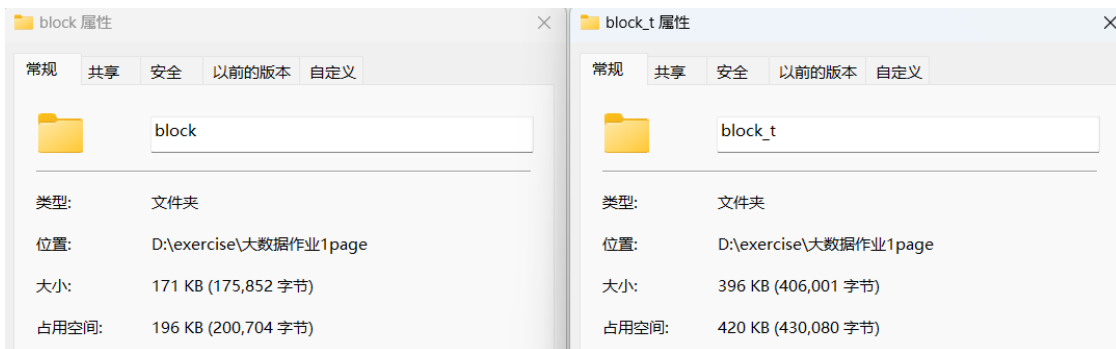


以上为读取文件并进行相应转化的代码。

为了比较二进制处理以及文本文件处理的性能差异，这里先放入一个分块算法与文本文件进行对比的小 demo，后续会进行更加详细的性能分析：

```
#文本文件处理的函数调用
c_index,score=block_stride_pagerank("Data.txt",0.85,1e-10,1000,100,
[data_block_stride_solve_t,data_decode_t])
#二进制处理的函数调用
c_index,score=block_stride_pagerank("Data.txt",0.85,1e-10,1000,100,
[data_block_stride_solve,data_decode])
```

我们通过函数指针的方式来使用不同的处理函数，以上为块算法中将块数量设置为 100，的分块 pagerank 算法。



左边为二进制文件夹，右边为文本文件夹，我们首先从数据大小就可以看出二进制处理后的文件大小是文本文件大小的约 1/2,这表明我们的压缩方法确实起到了作用。

然后我们简单比较一下两者的用时：

```
if __name__ == "__main__":
    c_index,score=block_stride_pagerank("Data.txt",0.85,1e-10,1000,100,[data_block_stride_solve,data_decode])
    write_back_result(c_index,score,"block_stride_result.txt")
    c_index,score=block_stride_pagerank("Data.txt",0.85,1e-10,1000,100,[data_block_stride_solve_t,data_decode_t])
    write_back_result(c_index,score,"block_stride_result.txt")

100
time: 11.101438999176025s.
100
time: 10.585489511489868s.
```

上面数据为二进制文件，下面的数据为文本文件处理，我们发现两者的时间用时并无明显差异。个人理解是，虽然二进制文件的读写时间要小于文本文件，但是将二进制数据转化为我们需要的数据这个用时是要大于文本文件转化的用时的。

## 2. dead ends 和 spider trap 节点处理：

首先对于 dead ends 节点的定义为：没有任何出链的节点。这会导致转移概率矩阵对应列的元素和为 0，在这种情况下继续进行迭代操作该节点的 PR 值逐渐变

为 0。我们小组选择将 dead ends 节点形成的转移矩阵的列都使用一个权重平均值去补齐，这也很好理解，一个网页不指向任何其他网页，同样也可以表示为该网页对于所有网页的 PR 贡献值相同，但其实我们可能会产生贡献值相同并不等价于没有贡献值的想法，我个人理解这是一种为了完成收敛的一种"折衷"。

之后对于 spider trap 节点的定义为：存在出链仅指向自身的节点。这会导致在迭代过程中网站的权重会向 spider trap 节点偏移置 1，这是与实际情况不符的。所以，我们采用随机游走的方案，其内容为用户在选择浏览网页的时候有  $\beta$  的概率会跟随出链打开网页，有  $1 - \beta$  的概率会随机打开其他网页，这也是大作业说明文件中要求我们说明  $\beta = 0.85$  的排名情况中的参数。

结合以上两点，我们可以将普通的 pagerank 算法的计算公式更改如下：

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

(其中  $r_i$  为  $PR_i$ ,  $d_i$  为节点  $i$  的出度,  $N$  为节点的数量)

结合以上公式我们便可以对于 pagerank 的基本代码进行实现：

```
def base_pagerank(file,data,belta,error,max_iteration):
    nodes=data_solve(file, 'data.bin')
    M=data_decode('data.bin')
    #只考虑有出度或是入度的节点
    s=set(sorted(sorted(M)+sorted(nodes)))
    s1=list(s)
    idx={}
    for i in range(len(s1)):
        idx[s1[i]]=i
    N=len(set(sorted(M)+sorted(nodes)))
    start = time.time()
    #使用Teleport解决Dead Ends问题
    a=np.zeros(N)
    for j,i in zip(range(N),s):
        if nodes[i]==0:
            a[j]=1/N
    r=np.ones(N)/N
    r_new=np.ones(N)
    temp2=(1-belta)/N
    #迭代直至收敛
    iteration=0
    while True:
        iteration+=1
        for i,h in zip(s,range(N)):
            m=np.zeros(N)
            temp1=np.dot(a,r)
```

```

        if len(M[i])!=0:
            for j in M[i]:
                temp1+=r[idx[j]]/nodes[j]
            temp1=beta*temp1
            r_new[h]=temp1+temp2
        if sum(abs(r_new-r))<error or iteration==max_iteration:
            break
        r=r_new.copy()
    end = time.time()
    print(iteration)
    print('time: {}s.'.format(end - start))
    index=np.argsort(r_new)[::-1][:100]
    c_index=[s1[i] for i in index]
    score=np.sort(r_new)[::-1][:100]
    return c_index,score

```

以上代码基本上是按照公式表述进行实现，先将数据进行读取，并进行相应处理，迭代方面通过循环得到其中每一个 $r_i$ 的值,其中对应收敛的判断条件为 $sum(r_i - r_i(old)) < error$ 。

这里放入一个函数的调用，表明最终程序已迭代至收敛：

```

if __name__ == "__main__":
    c_index,score=base_pagerank('Data.txt', 'data.bin',0.85,1e-10,1000)
    write_back_result(c_index,score,"base_result.txt")

```

```

100
time: 7.5937066078186035s.

```

此函数表明，我们按照要求将 $\beta$ 设置为 0.85，精度为 1e-10，最大迭代次数为 1000 次，这里输出的 100 表明最后一次迭代的次数为 100 次，表明函数在 100 次时就已经达到精度要求（即收敛）。

### 3. 实现分块计算：

#### 3.1. 分块方法：

按照区间范围进行分块的思想，数据处理后的稀疏矩阵的索引是递增的，所以我们采取的分块策略为通过分块数量以及节点最大索引来确定分块的范围：

$$width = \frac{max\_index}{block\_number}$$

(其中 $width$ 为每一块的区间， $max\_index$ 为最大的索引 $id$ ， $block\_number$ 为分块的数量)

有了分块的范围我们即可以，将稀疏矩阵分解为多个块，每个块由索引范围内节点行向量组成。这样就解决了稀疏矩阵过大无法完整的存储在内存的问题，但是会多次的涉及文件读写，在时间上会产生额外的开销。

### 3.2. 代码讲解：

```
#二进文件的数据处理优化
def data_block_stride_solve(file,block_number):
    m = defaultdict(list)
    n = defaultdict(int)
    #得到节点出度，以及知道目的节点由哪些节点指向
    with open(file, 'r') as f:
        for i in f:
            a,b = [int(x) for x in i.split()]
            n[a]+= 1
            m[b].append(a)

    m=dict(sorted(m.items()))
    m1=copy.deepcopy(m)
    #每一块的区间宽度
    block_width=int(max(m)/block_number)
    lift=0
    right=-1
    l=list()
    r=list()
    filepath="./block"
    if not os.path.exists(filepath):
        os.mkdir(filepath)
    else:
        shutil.rmtree(filepath)
        os.mkdir(filepath)

    for i in range(block_number):
        lift=right+1
        l.append(lift)
        if i!=block_number-1:
            right=right+block_width
        else:
            right=max(m)
        r.append(right)
        block_name="./block/block_"+str(i)
        with open(block_name,'wb') as f:
            for a,b in sorted(m.items()):
                if a>=lift and a<=right:
                    x=a.to_bytes(2,"little")
```



```

        h=len(b)
        x=x+h.to_bytes(2,"little")
        for i in b:
            x=x+i.to_bytes(2,"little")
        f.write(x)
        del(m[a])
    else:
        break
for i in m1.keys():
    if n[i]==0:
        continue

return n,l,r

```

首先我们的数据处理函数会根据块数量计算出每一块的范围，范围存储在 **l,r** 列表中，在以后的计算中会用到。之后我们对于每一个节点根据其节点的索引号存储在对应范围的块中，存在 **block** 文件夹中如下图所示：

📁 / 大数据作业1page / block /

Name	Last Modified
📄 block_0	an hour ago
📄 block_1	an hour ago
📄 block_10	an hour ago
📄 block_11	an hour ago
📄 block_12	an hour ago
📄 block_13	an hour ago
📄 block_14	an hour ago
📄 block_15	an hour ago
📄 block_16	an hour ago
📄 block_17	an hour ago
📄 block_18	an hour ago
📄 block_19	an hour ago
📄 block_2	an hour ago
📄 block_20	an hour ago
📄 block_21	an hour ago
📄 block_22	an hour ago
📄 block_23	an hour ago
📄 block_24	an hour ago
📄 block_25	an hour ago
📄 block_26	an hour ago



```

def
block_stride_pagerank(file,belta,error,max_iteration,block_number,data_func):
    nodes,ll,rr=data_func[0]("Data.txt",block_number)
    start = time.time()
    N=len(nodes)
    s=list(sorted(nodes))
    idx={}
    for i in range(len(s)):
        idx[s[i]]=i
    a=np.zeros(N)
    for j,i in zip(range(N),s):
        if nodes[i]==0:
            a[j]=1/N
    r=np.ones(N)/N
    r_new=np.ones(N)
    temp2=(1-belta)/N
    iteration=0
    while True:
        block_index=0
        M=data_func[1](block_index)
        iteration+=1
        for i,h in zip(s,range(N)):
            if i>=ll[block_index] and i<=rr[block_index]:
                m=np.zeros(N)
                temp1=np.dot(a,r)
                if len(M[i])!=0:
                    for j in M[i]:
                        temp1+=r[idx[j]]/nodes[j]
                temp1=belta*temp1
                r_new[h]=temp1+temp2
            else:
                block_index+=1
                M=data_func[1](block_index)
                m=np.zeros(N)
                temp1=np.dot(a,r)
                if len(M[i])!=0:
                    for j in M[i]:
                        temp1+=r[idx[j]]/nodes[j]
                temp1=belta*temp1
                r_new[h]=temp1+temp2
        if sum(abs(r_new-r))<error or iteration==max_iteration:
            break
        r=r_new.copy()
    end = time.time()
    print(iteration)

```

```

print('time: {}'.format(end - start))
index=np.argsort(r_new)[::-1][:100]
c_index=[s[i] for i in index]
score=np.sort(r_new)[::-1][:100]
return c_index,score

```

之后我们计算更新 $r_i$ 的时候，只需首先判断该节点索引是否在当前块范围中，若在，则进行分数的计算，反之则读取下一个块进行相应的计算。

### 3.3. 结果展示和比较：

```

: if __name__ == "__main__":
    c_index,score=block_stride_pagerank("Data.txt",0.85,1e-10,1000,100,[data_block_stride_solve,data_decode])
    write_back_result(c_index,score,"block_stride_result.txt")

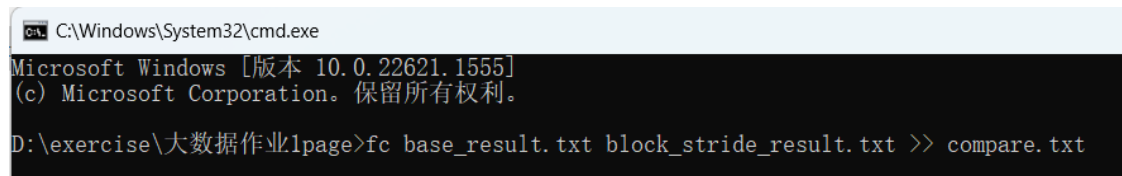
```

```

100
time: 11.476598024368286s.

```

在 100 次迭代的时候到达了精度要求（收敛），我们对比分块算法与基础算法的结果文件：

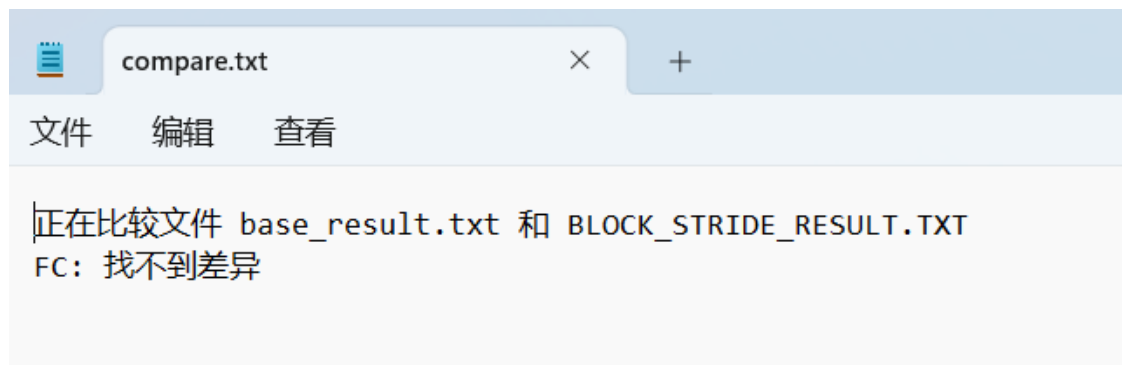


```

C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.22621.1555]
(c) Microsoft Corporation。保留所有权利。

D:\exercise\大数据作业1\page>fc base_result.txt block_stride_result.txt >> compare.txt

```



compare.txt

文件 编辑 查看

正在比较文件 base\_result.txt 和 BLOCK\_STRIDE\_RESULT.TXT  
FC: 找不到差异

我们通过 windows 命令行的方式将两个结果文件进行比较，两个文件结果一致。

4. 正确性验证:

4.1.与调用 network 库比较

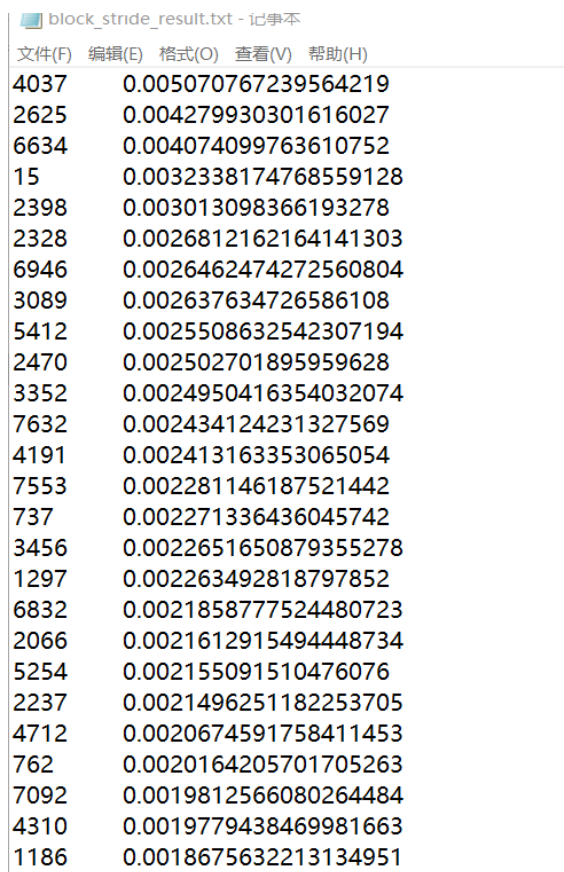
networkx_result.txt	
文件	编辑 查看
4037	0.004556412278710509
2625	0.003845958019340682
6634	0.003721934823415891
15	0.0031553234260584245
2398	0.00267108237013648
2328	0.002617131253537157
2470	0.0023842205688539933
5412	0.0023820335066259053
7632	0.002281121461848156
3089	0.0022616729833260453
3352	0.002229801546922114
737	0.0021832425895811215
4191	0.0021493827108931725
2237	0.0021434131260656742
3456	0.0021400983232251923
5254	0.0021179307592779973
6832	0.0020839077374494534
7553	0.0020786639036499766
2066	0.002024260606407244
1297	0.0019830055241318355
7092	0.0018967378360034738
4310	0.0018744344352668437
4712	0.001817040504943371
6774	0.0017859355361002302
762	0.0017738808796386848
6946	0.0017721108786300356

block_stride_result.txt - 记事本	
文件(F)	编辑(E) 格式(O) 查看(V) 帮助(H)
4037	0.004989267502499286
2625	0.004070535050564411
6634	0.003726160656617783
15	0.0030983201054967115
2398	0.0028149269710186875
2328	0.0025752685483611433
2470	0.0025435497071087144
3089	0.0024706916979130703
6946	0.0023736645363846296
3352	0.0023630920576154335
5412	0.0023452885687513165
4191	0.0022922343237360542
7632	0.0022669196836236317
7553	0.0021809973239990386
737	0.002143827076727219
1297	0.002136097146944484
3456	0.0021100827158282006
2237	0.0021062131151581235
5254	0.0020782291638185893
6832	0.0020708169902558834
2066	0.001999589740554574
4712	0.0019038728975277132
762	0.0018815137109828037
7092	0.001878313120806457
1186	0.0018647036140238195
4310	0.0018579581248575833

我们通过我们的结果文件与调库 `network` 产生结果文件进行比较，发现排名大体相同，但分数存在些许差异，差异原因可能是我们的算法在具体实现上和 `network` 中有些许不同。

#### 4.2.与 $\beta$ 值分别设为 0.80 和 0.90 相比较

将 $\beta$ 值设为 0.90:



4037	0.005070767239564219
2625	0.004279930301616027
6634	0.004074099763610752
15	0.0032338174768559128
2398	0.003013098366193278
2328	0.0026812162164141303
6946	0.0026462474272560804
3089	0.002637634726586108
5412	0.0025508632542307194
2470	0.002502701895959628
3352	0.0024950416354032074
7632	0.002434124231327569
4191	0.002413163353065054
7553	0.002281146187521442
737	0.002271336436045742
3456	0.0022651650879355278
1297	0.002263492818797852
6832	0.0021858777524480723
2066	0.0021612915494448734
5254	0.002155091510476076
2237	0.0021496251182253705
4712	0.0020674591758411453
762	0.0020164205701705263
7092	0.0019812566080264484
4310	0.0019779438469981663
1186	0.0018675632213134951

我们会发现排名基本相同，但分数会提高一些，我们推断这主要是源于 $\beta$ 值提高，用户随机打开网页的概率减小，所以按马尔科夫链进行网页间的跳转，导致与其他网页关联较多的网页更容易被跳转到，所以分数也会更高。

将 $\beta$ 值设为 0.80:

block\_stride\_result.txt - 记事本

文件(F)	编辑(E)	格式(O)	查看(V)	帮助(H)
4037	0.004884304844885686			
2625	0.0038572080173298008			
6634	0.003406449124566278			
15	0.0029571351746511803			
2398	0.0026235110197099353			
2470	0.002560592923635975			
2328	0.0024643675096333285			
3089	0.0023057272838044054			
3352	0.002231039123416633			
4191	0.0021713700302950145			
5412	0.00214766091727928			
6946	0.002127369724456567			
7632	0.002104810477620287			
7553	0.0020727842511760065			
2237	0.002052961920238048			
737	0.002015491158530593			
1297	0.0020068703043267778			
5254	0.001997253908884564			
3456	0.001959626375245605			
6832	0.001953679903044848			
1186	0.0018485446734593895			
2066	0.0018433665596952198			
6774	0.00178278260172526			
7092	0.0017747886533352516			
4712	0.0017497517260702485			
762	0.0017473823729742405			
4310	0.0017395524686587478			

我们会发现也是排名基本相同，但分数会降低一些，我们推断这主要是源于 $\beta$ 值降低，用户随机打开网页的概率增加，所以更容易跳到无关网页中，相比热门的网站被浏览的概率会降低，所以分数也会降低。

## 五. 实验感想

通过本实验，我们深刻了解了 PageRank 算法的实验原理和具体实现。PageRank 是一种对网页重要性进行客观、公正评价的算法。它不直接计算链接的数量，而是将从网页 A 指向网页 B 的链接视作由网页 A 给网页 B 投出的一票，这一票的分量也取决于 A 本身的重要性。PageRank 从而根据网页 B 收到的综合投票数来评估网页的重要性。以上这些决定了谷歌搜索结果的相关性和可靠性。

此外，在本实验中，我们对 PageRank 的需要解决的问题进行了深入讨论，包括 dead ends 和 spider traps 问题、稀疏矩阵优化和分块矩阵等，并在代码部分针对这些问题进行了实际解决。总之，通过本次实验，我们对大数据时代的搜索引擎技术的理解有了更深一步的了解。