



南開大學
Nankai University

计算机与网络空间安全学院
大数据计算及应用期中报告

PageRank 算法设计与实现

姓名：杨鑫 吴晨宇

学号：2011028 2012023

专业：信息安全 信息安全

2023 年 4 月 30 日

目录

1 概述	3
1.1 PageRank 算法背景	3
1.2 PageRank 算法原理	3
1.3 PageRank 算法问题及优化	5
1.3.1 优化稀疏矩阵	5
1.3.2 Dead Ends 问题	6
1.3.3 Spider Traps 问题	7
1.3.4 矩阵分块计算	8
1.3.5 迭代收敛问题	9
2 实验概述	9
2.1 实验要求	9
2.2 实验思路	10
3 实验环境	10
3.1 开发语言环境	10
3.2 Windows 端系统环境	10
4 数据集分析	10
4.1 数据集说明	10
4.2 数据编号	11
5 基础算法代码解析	11
6 优化算法代码解析	12
6.1 优化算法	12
6.1.1 稀疏矩阵的存储优化	12
6.1.2 分块计算的优化方法	13
6.1.3 分块 PageRank 算法	13
6.1.4 PageRank 类	13
6.1.5 PageRank 类成员函数介绍	15
6.2 并行优化	21
7 运行过程	22
7.1 普通算法	22
7.2 分块优化算法	22
7.2.1 并行优化算法	24
8 实验结果分析	24
8.1 teleport 参数 β 对程序运行结果的影响	24
8.1.1 迭代终止条件 ϵ 对程序运行结果的影响	25
8.2 分块个数对程序运行结果的影响	26
8.3 并行线程对于程序运行结果的影响	26

9 小组分工	27
10 实验总结	28
11 代码仓库	28

1 概述

1.1 PageRank 算法背景

在互联网应用的早期阶段，搜索引擎采用分类目录的方法，通过人工进行网页分类，并整理出高质量的网页。但是后来随着网页的逐渐增多，人工分类已经不现实，这个时期的搜索引擎采用文本检索的方法，即计算用户检索的关键词与网页内容的相关度，返回所有结果，但关键词并不能反映网页的质量，搜索效果不好。

PageRank 算法是一种用于评估网页重要性的算法，是 Google 搜索引擎的核心技术之一。PageRank 算法最初由 Google 公司的创始人之一拉里·佩奇和谢尔盖·布林在 1998 年提出。他们认为，网页的重要性应该与其他网页对其的引用数量有关，同时还应考虑这些引用网页的重要性。因此，他们开发了 PageRank 算法，该算法通过对网页间的链接关系进行分析来计算每个网页的重要性。

1.2 PageRank 算法原理

PageRank 算法的核心思想是：一个网页的重要性取决于其被其他重要网页所链接的数量和质量。具体来说，如果一个网页被很多其他网页所链接，那么它就更有可能被用户所访问，从而具有更高的重要性。而如果这些链接来自于其他重要的网页，那么这个网页的重要性就更高。下图可以反映一个网页节点重要度的大小不仅取决于链接向它的网页节点的数量，还取决于链接向它的网页节点的重要度大小。如图1.1所示

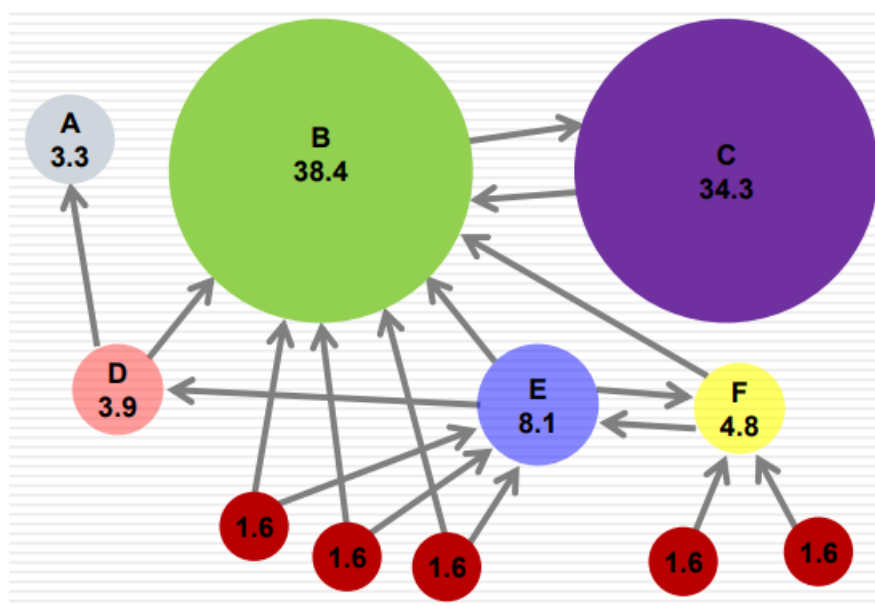


图 1.1: 网页节点重要度示意图

基于上述，PageRank 算法通过将网页之间的链接关系表示为一个图来计算网页的重要性。在这个图中，每个网页都表示为一个节点，每个链接则表示为一条有向边。如果网页 A 链接到网页 B，那么就在节点 A 和节点 B 之间连一条有向边。在计算过程中，PageRank 算法将每个节点的初始权重设置为 1。然后，它通过迭代计算每个节点的 PageRank 值，直到收敛为止。具体来说，每次迭代时，PageRank 算法会将每个节点的 PageRank 值更新为其所有入度节点的 PageRank 值之和的加权平均值。其中，权重是入度节点的出度节点数量的倒数。这个过程可以用以下公式表示：

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

其中, r_j 表示节点 j 的 PageRank 值, r_i 所有链接向该节点的页面的当前 PageRank 值, β 是一个阻尼因子 (通常取值为 0.85), N 是图中节点数量, d_i 表示节点 i 的出链个数。

使用下图以便更好的理解 PageRank 算法的思想, 如图1.2所示

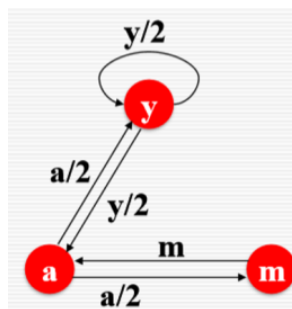


图 1.2: PageRank 算法思想

从该图中可以得到三个节点的重要度计算公式:

$$r_y = \frac{r_y}{2} + \frac{r_a}{2}$$

$$r_a = \frac{r_y}{2} + r_m$$

$$r_m = r_a/2$$

由上图可知每个页面都将自己的一部分 r 值传递给其他页面, 我们可以通过传递给某个页面的所有 r 值进行求和来计算出它的 r 值。在 PageRank 算法初始的执行过程中, 我们可以给每个页面赋予一个初始 r 值—— $\frac{1}{N}$, 其中 N 为节点的总数, 然后通过迭代计算得到该每个节点的 r 值。迭代计算的停止条件为所有页面节点的 r 值与旧的所有页面节点的 r 值之间的差值小于一个预先设定的值 ϵ , 即:

$$|r^{(t+1)} - r^{(t)}|_1 < \epsilon$$

其中 $|x|_1 = \sum_{i \rightarrow N} |x|_1$ 是 L_1 范式。

在实际过程中, 我们可以使用二维矩阵 M 来迭代更新每个页面的 r 值。 M 矩阵的生成方法是: 如果节点 i 指向 j , 那么 $M_{ji} = \frac{1}{d_i}$, 否则 $M_{ji} = 0$, 一般来说 M 的矩阵的每一列和为 1。有了矩阵 M , PageRank 的迭代算法就可以表示为下面的形式:

$$A = \beta M + (1 - \beta) \left[\frac{1}{N} \right]_{N \times N}$$

其中 $\left[\frac{1}{N} \right]_{N \times N}$ 是 $N \times N$ 的矩阵且每一个元素的值都是 $\frac{1}{N}$ 。因此每次迭代只需要计算:

$$r = A \cdot r$$

PageRank 算法的优点是能够有效地评估网页的重要性, 并且不容易被人工干扰。但是, 它也存在一些缺点, 例如容易被滥用和操纵。因此, 在实际应用中, 需要结合其他算法和人工干预来提高搜索结果的质量。

1.3 PageRank 算法问题及优化

1.3.1 优化稀疏矩阵

在真实 PageRank 算法应用场景中一般给出的数据集都是及其稀疏矩阵的形式。假设数据集中一共包含 N 个节点，那么不做优化的矩阵大小即为 $N \times N$ ，但是每个节点的出度数相比于 N 来说很小，因此将会存在很大的空间不会被使用。在实际中可能是基于“每个网页平均有 10 个外向链接”的情况，因此我们并不需要 $N \times N$ 大小的空间来存储矩阵 M ，相反， $10 \times N$ 的空间就已经足够了。因此为了减少系统运行的空间消耗，这里可以使用链表数据结构形式来存储矩阵 M ，链表中的每个元素包含了数据集中的源节点、该节点的出度、以及它所指向的目的节点。使用链表矩阵 M 进行一次迭代的算法：遍历整个三元组链表，每次从三元组链表中取出一个元素，然后遍历三元组中指向目的节点的链表中的网页编号 j ，对每个 $R_{new}[j]$ 做：

$$r^{new}(dest_j) += \beta \frac{r^{old}(i)}{d_i}$$

其中 r^{new} 是初始值全为零的列向量。这个过程相当于实现了：

$$r^{new}[j] = \sum_{i \rightarrow j} \frac{r^{old}[i]}{d_i}$$

具体的算法流程如图1.3所示

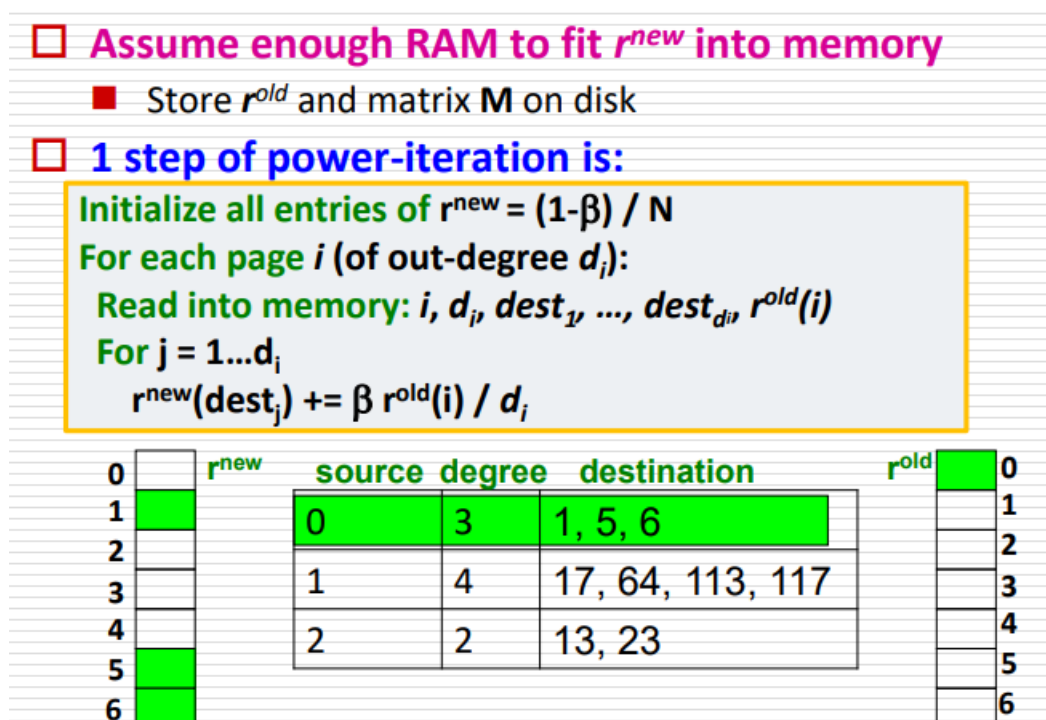


图 1.3: 稀疏矩阵优化示意图

1.3.2 Dead Ends 问题

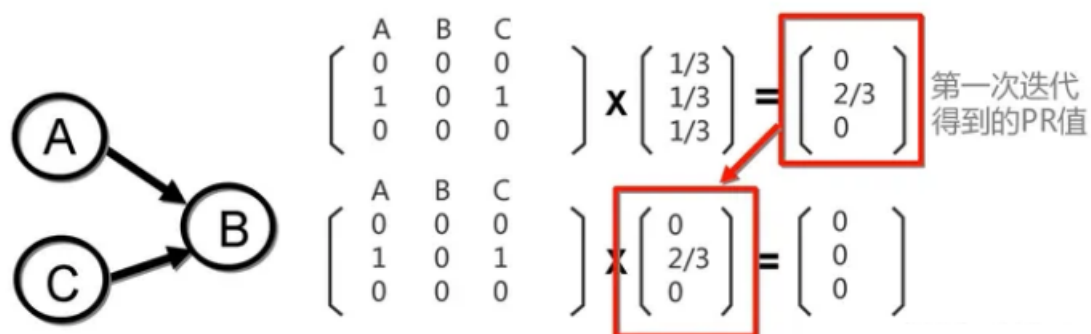


图 1.4: Dead Ends 示意图

如图1.4所示, B 没有任何出链, 这时候矩阵 M 的该列为 0, 这种节点会逐渐吞噬掉整个数据集的能量, 经过多次循环之后, 会导致网站的权重变为 0。此时数据集组成的图不是强连通的, 即存在某一类节点不指向其他节点, 这种情况下我们的 PageRank 算法就不满足收敛性了。

解决方案是通过使用 teleport, 即假设每个页面有很小的概率拥有一个指向其他页面的链接, 在每次的计算中, 以 β 的概率按网络的真实情况走, 再以 $(1 - \beta)$ 的概率平均的走向其他所有节点。这样将节点图转换成列转移概率矩阵, 再修正马尔可夫矩阵 M :

$$M \leftarrow M + a^T \left(\frac{ee^T}{n} \right)$$

其中 $a = a[a_0, a_1, \dots, a_n](a_i)$, 当修正前的 M 中第 i 列数值全为 0 (即对应节点无出链) 时, 则 a_i 中的元素全为 1, 否则矩阵 a_i 中元素全为 0。 ee^T 为由元素 1 填满的 $n \times n$ 矩阵, n 为 M 矩阵的行数 (或列数)。如图1.5所示

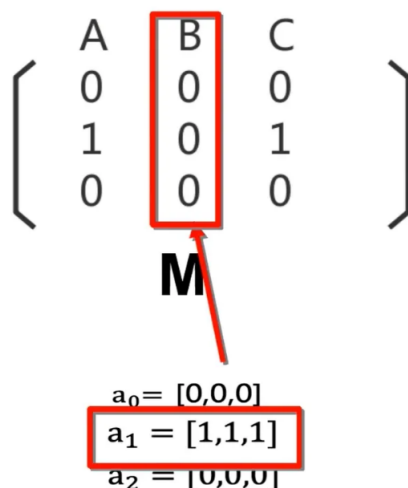
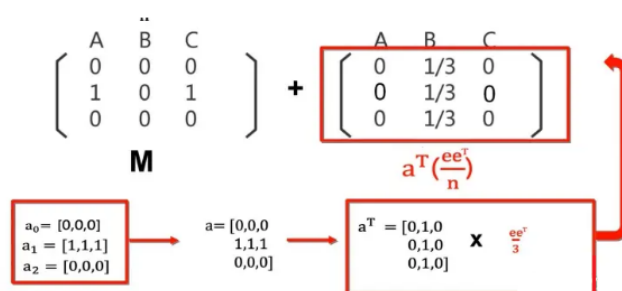


图 1.5: Dead Ends 修正方案

如上图, 因为第一列元素全部为 0, 所以有 $a_1 = [1, 1, 1]$, 则第 0 列和第 2 列不全为 0, 所以 a_0, a_2 元素全为 0。

图 1.6: 对 M 矩阵进行修正过程

对 M 矩阵进行修正的过程如上图1.6所示。

1.3.3 Spider Traps 问题

按照这个规律，我们在多次循环之后，会发现这个模型中A的PR值都会归于1，其他归为0

PR 值	PR (A)	PR (B)	PR (C)
循环次数 i			
$i = 0$ PR值初始化 = $1/N$	$1/3$	$1/3$	$1/3$
$i=1$	$2/3$	$1/6$	$1/6$
$i=2$	$5/6$	$1/12$	$1/12$
$i=3$	$11/12$	$1/24$	$1/24$
$i=n$	1	0	0

图 1.7: Spider Traps 问题

A 节点与其他节点之间不存在出链 (更具体地说，节点 A 存在闭环)，此为 Spider Traps 问题。如上图1.7所示，其会导致网站权重变为向某一个含自环的节点偏移 (因为此节点 PR 值最终会趋于 1)。

解决方案是通过使用 Random Teleport，将节点图转换为列转移概率矩阵，再修正马尔可夫矩阵 M (随机浏览模型)。

$$M \leftarrow \beta M + (1 - \beta) \frac{ee^T}{n}$$

β 为跟随出链打开网页的概率， $(1 - \beta)$ 为随机跳转到其他网页的概率，例如浏览网页 A 的时候有一定的概率打开网页 B 或 C。 ee^T 为由元素 1 填满的 $n \times n$ 矩阵， n 为 M 矩阵的行数 (或列数)。具体过程如图1.8所示

$$M = \beta \begin{bmatrix} & A & B & C \\ 1 & 1/2 & 1/2 \\ 0 & 0 & 1/2 \\ 0 & 1/2 & 0 \end{bmatrix} + (1 - \beta) \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

$$M = \beta M + (1 - \beta) \frac{ee^T}{n}$$

图 1.8: Spider Ends 解决方案

β 的取值范围一般在 $[0.8, 0.9]$ 。修正后的 M 矩阵中，列中的元素值会发生变化，但是每一列的求和结果仍是 1 保持不变。

在使用 β 解决 Spider Traps 问题的时候不能顺便解决 Dead Ends 问题，这是因为在使用 β 解决 Dead Ends 问题时，修正后的 M 不满足转移概率矩阵的性质——列之和为 1。因此只用 β 是不足以解决 Dead Ends 问题的。

因此最终的修正公式如下，本次实验也是基于此公式进行 PageRank 的计算：

$$r^{new} = [\beta(M + a^T(\frac{ee^T}{n})) + (1 - \beta)\frac{ee^T}{n}] * r^{old}$$

其中 r^{new} 表示所有节点 PageRank 值组成的向量， r^{old} 表示更新前的 PageRank 值组成的向量，初始化为 $\frac{1}{N}$ 。

1.3.4 矩阵分块计算

尽管已经用链表来优化了稀疏矩阵减少了空间损耗，但是相对于庞大的数据集来说计算还是十分困难的。例如假设数据集中一共包含 N 个节点，那么矩阵 M 的大小为 $10 \times N$ ，当数据集中存在 10 亿个网页的时候，假设每个节点的信息至少使用 8 个字节存储，那么其空间消耗就是 $10 \times 10^8 \times 8 = 80GB$ 。

此数据量的大小我们是不可能一次性将其全部读入到内存中进行处理、计算。假如内存小到连本次迭代的 r^{new} 都存不下，那么解决方案就是将 r^{new} 分为几块向量，在这种情况下， r^{new} 被分成几块，每次迭代就要从磁盘中读取多少次整个矩阵 M 和 r^{old} ，导致一次迭代计算的实践复杂度大幅增加。

因此可以采用 Block-Stripe Update Algorithm，该算法把 M 打散，在每一次计算某个节点下一时刻的 r 值时，只需将链表中目标节点包含当前节点的链表元素项加载到内存中，因为只有这些项决定了该节点的 r 值。这种做法使得整个迭代过程对矩阵 M 的读取次数接近为 1(因为打散后的矩阵 M 比不打散的矩阵 M 要大)，这样节省了很大的空间开销。本次实验采用的分块方法也是基于 Block-Stripe Update Algorithm，这里因为节点的个数比较小因此可以将节点的整个 PageRank 向量放入内存，然后分别与读入的矩阵分块进行计算，然后得到分块的 r^{new} ，组合起来就是新的 PageRank 值。算法示意图如图 1.9 所示

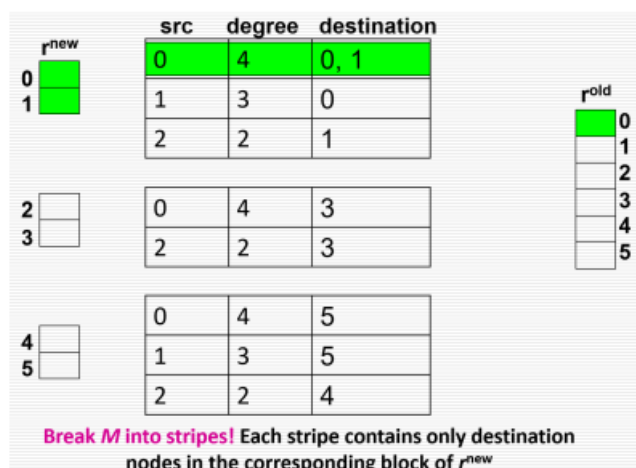


图 1.9: Block-Stripe Update Algorithm 示意图

1.3.5 迭代收敛问题

对于上述的 PageRank 算法为什么能收敛的问题，需要转换为 Markov 过程。定义矩阵

$$A = \beta M + (1 - \beta) \left[\frac{1}{N} \right]_{N \times N}$$

则 PageRank 的计算过程可以转化为：

$$r^{new} = A \cdot r^{old}$$

于是该过程就转为了一个 Markov 过程了，下面要证明 PageRank 收敛，即证明其满足 Markov 过程收敛的条件：

- A 为随机矩阵 (A 矩阵所有元素都大于等于 0，并且每一列的元素和都为 1)；
- A 为不可约的 (当图是强连通时，A 为不可约，我们之前定义各个网页都是可相互转跳的)；
- A 为非周期的。

以上条件均满足因此 PageRank 是收敛的，且与初始值无关。

2 实验概述

2.1 实验要求

基于给定数据集计算 PageRank：

- 语言: C/C++/JAVA/Python
- 考虑 dead ends 和 spider trap 节点
- 优化稀疏矩阵
- 实现分块计算
- 程序需要迭代至收敛

- 不可直接调用接口，例如实现 pagerank 时，调用 Python 的 networkx 包
- 结果格式 (.txt 文件): [NodeID] [Score]

2.2 实验思路

在本次实验中，本组首先对 PageRank 的基本算法进行了实现，然后基于基本算法进行对稀疏矩阵分块的处理、Dead ends 的处理、对 Spider trap 的处理的实现。

然后本组在基于上述实现后对实验过程中进行火焰图和性能分析，然后针对计算量较大的代码部分进行并行优化，并在 windows 平台上进行结果、性能分析。

最后基于提供的数据集生成结果集，并针对参数例如 β 等进行不同设置得到其对结果的影响，比较不同参数情况下 PageRank 算法的评分结果。

3 实验环境

3.1 开发语言环境

- 开发语言: C++11
- 编译器: g++(i686-posix-sjlj) 4.9.2

3.2 Windows 端系统环境

- 操作系统: Windows 10 家庭中文版 (64 位)
- 硬件环境: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz; RAM 16.0GB

4 数据集分析

4.1 数据集说明

在原始数据集 Data.txt 中，每一行包含两个节点 ID：源节点和目的节点，表示存在一条从源节点指向目的节点的边。示意图如图4.10所示



图 4.10: 原始数据集 Data.txt 的格式分析

经过程序统计，在原始数据集中，出现最小节点 ID 为 3，最大节点为 8297，在节点 ID 从 0 到 8297 的这些节点中共有 6262 个 ID 存在出链，一共存在 83852 条边，而从 0 到 8297 节点中没有出现过的节点 ID 是既不存在出链也不存在入链，在计算过程中会对 ID 重新进行编号，因此这些将不会被考虑进去，最后生成的结果文件 Result.txt 只有 6262 个节点的信息。

4.2 数据编号

在上述数据集说明的地方我们可以看到有些 ID 并没有被使用，因此对于数据集中的 ID 是离散的，而在 PageRank 计算过程中，PageRank 值是以链表的方式存储的是顺序存储方式，因此如果是离散的不仅对空间有所浪费，而且还是增加时间消耗。因此最好的办法是将离散的 ID 映射到一个有序的链表上，本次实验中就是将离散的 ID 按照 ID 大小重新映射到另一个链表上顺序编号，这样在计算过程中就能减少不少的麻烦，最后根据映射的关系表得到原始的节点 ID。示意图如图4.11所示

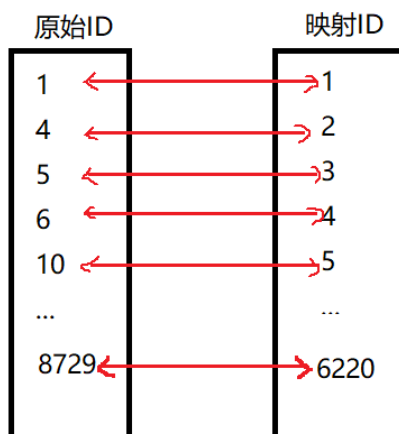


图 4.11: 映射关系示意图

5 基础算法代码解析

基础算法代码在运行时，首先不分块的读入所有链接的边的数据，使用邻接链表进行存储，并记录边数和节点数等信息；然后，使用 PageRank 算法进行排名计算，计算的核心公式为：

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

其中， r_j 表示节点 j 的 PageRank 值， r_i 所有链接向该节点的页面的当前 PageRank 值， β 是 teleport 参数（通常取值为 0.85）， N 是图中节点数量， d_i 表示节点 i 的出链个数。

在进行计算时，teleport 参数 β 设置为 0.85，并且需要设定一个迭代终止条件的精度 Epsilon，在迭代到根据公式更新时的误差 loss 小于该精度时，若干轮次迭代结束；最终，使用 sort 函数进行排序，将排名为前 100 的 NodeID 与它们对应的分数 score 进行输出。

具体 PageRank 的过程为先根据入度遍历邻接表，把所有存在出度的边找出并计算 $\beta \frac{r_i}{d_i}$ ，便计算

边累加，在遍历完所有存在出度的边之后，将其全部求和，即得到核心计算公式中的部分内容：

$$\sum_{i \rightarrow j} \beta \frac{r_i}{d_i}$$

最后再计算 $r_i = (1 - \beta) \frac{1}{N}$ 部分即可得到心得 Rank 值；然后再将新旧 Rank 值相减取绝对值，即为 loss 值，当 loss 值小于迭代终止条件的精度 Epsilon 时，即更新后新旧 Rank 值变化不大，这就说明算法收敛了，所以算法可以停止了。

基础算法核心代码

```

1  double loss;
2  do
3  {
4      loss = 0;
5      for (auto i : E_in)
6      {
7          Rn[i.first] = 0;
8          for (int j = 0; j < i.second.size(); j++)
9              if (out_dgree[i.second[j]] != 0)
10                 Rn[i.first] += beta * (Ro[i.second[j]] / out_dgree[i.second[j]]);
11             else
12                 Rn[i.first] += 0;
13         }
14         double sum0 = 0;
15         for (auto i : Rn)
16             sum0 += i;
17         for (int i = 0; i < Rn.size(); i++)
18         {
19             Rn[i] += (1 - sum0) / n;
20             loss += (Rn[i] > Ro[i]) ? (Rn[i] - Ro[i]) : (Ro[i] - Rn[i]);
21             Ro[i] = Rn[i];
22         }
23     }
24     while (loss > Epsilon);

```

6 优化算法代码解析

6.1 优化算法

6.1.1 稀疏矩阵的存储优化

这里存储稀疏矩阵的数据结构是 map+set 的方式，记录单个节点的所有出度节点，这种方式存储能尽可能的节省空间，避免其他冗余数据同时提供更快的索引。原理如图6.12所示

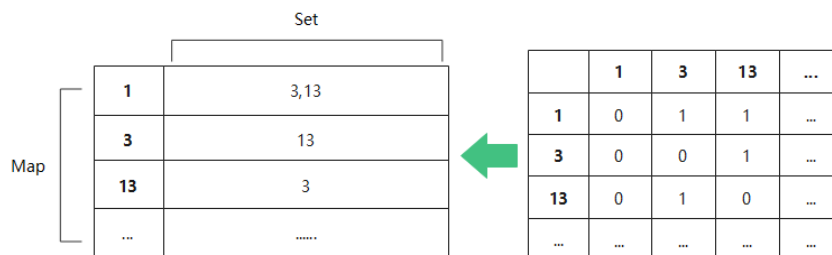
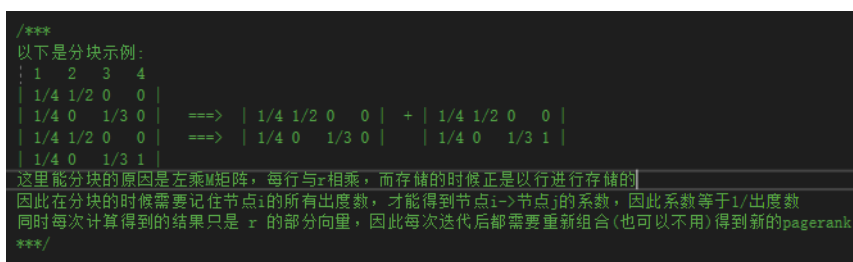


图 6.12: 稀疏矩阵存储数据结构示意

6.1.2 分块计算的优化方法

如果每次迭代的时候将整个矩阵左乘必然会导致很大的内存开销，需要将整个矩阵都读取到内存中进行计算，即使我们将稀疏矩阵的存储方式优化了，但是对于给定的数据集中存在上万个边，矩阵的读入对于内存是很大的开销，因此这里采用分块的思想，将矩阵分块几块进行分块计算，然后将每块的结果加起来组成最后的 pagerank 结果。原理如图6.13所示



6.1.3 分块 PageRank 算法

以下是分块计算 PageRank 的源码结构：

```

/blockPagerank
├── pageRank.h
├── pageRank.cpp
├── main.cpp
└── mem.h

```

其中 blockPagerank/pageRank.h 里面是 PageRank 类的定义，还有程序中使用的一些宏定义；blockPagerank/pageRank.cpp 是 PageRank 类成员函数的定义；blockPagerank/main.cpp 是程序入口；blockPagerank/mem.h 是程序中使用的其他人写好的测量进行内存的模块。

6.1.4 PageRank 类

为了使代码条理更加清晰简洁，我们将 PageRank 算法进行类的封装，设计了一个 PageRank 类并附上了相应的类定义，其中相关成员遍历和函数解释已经在注释中给出。

PageRank 类

```

1  class PageRank {
2  private:
3      map<int, set<int>> matrix;
4          // 存储稀疏矩阵, 使用map+set的方式, 能减少存储空间, 以及更快的索引
5      int maxIteration;          // 最大迭代轮数
6      long double beta;          // 随机游走系数
7      long double convergence;   // 收敛限界
8      map<int, int> nodeToIndex;
9          // 初始编号与离散化后编号的映射, 这里离散化是为了去除一些并不存在的节点(即没有入度也没有出度)
10     map<int, int> indexToNode;   // 离散化后的编号与初始编号的映射
11     vector<long double> pr;      // PageRank数组
12     vector<int> degree;          // 离散后每一个点的出度数
13     int totalNum;               // 离散化后的编号总数
14     int showNum;
15     int blockNum;              // 分块块数
16     int totalEdges;            // 数据总共的边数
17     int minItemNum, maxItemNum; // 最小编号和最大编号
18     vector<pair<long double, int>> output; // 保存最后排序结果
19
20 public:
21     long double getConvergence() const;
22     void setConvergence(long double convergence);
23     long double getBeta() const;
24     void setBeta(long double beta);
25     int getMaxIteration() const;
26     void setMaxIteration(int maxIteration);
27     int getMaxItemNum() const;
28     void setMaxItemNum(int maxItemNum);
29     int getMinItemNum() const;
30     void setMinItemNum(int maxItemNum);
31     int getBlockNum() const;
32     void setBlockNum(int blockNum);
33     int getShowNum() const;
34     void setShowNum(int showNum);
35
36     // 读取文件
37     void readFile(string dataFile);
38
39     // PageRank构造函数
40     PageRank(int _minItemNum = -1, int _maxItemNum = -1, int _maxIteration = -1, long
41         double _beta = 0,
42         int block_nums = 1, long double convergence = 1e-5, int _showNum = 0);
43
44     // 默认析构函数
45     ~PageRank() = default;
46
47     // 插入值到matrix中
48     void insertItem(int a, int b);
49
50     // pageRank算法
51     void pageRank();
52
53     // 读取文件得到最大值和最小值
54     void getItemRange(string dataFile);

```

```

47 //打印相关信息
48 void print();
49 };

```

6.1.5 PageRank 类成员函数介绍

首先 PageRank 的构造函数如下，仅仅用来进行初始化操作。

构造函数

```

1 PageRank::PageRank(int _minItemNum, int _maxItemNum, int _maxIteration, long double
   _beta, int _blockNum,
2 long double _convergence, int _showNum) : maxItemNum(_maxItemNum),
   minItemNum(_minItemNum), maxIteration(_maxIteration),
3 beta(_beta), blockNum(_blockNum), convergence(_convergence), showNum(_showNum) {
4 //清空matrix, nodeToIndex, indexToNode
5 this->matrix.clear();
6 this->nodeToIndex.clear();
7 this->indexToNode.clear();
8 this->totalEdges = 0;
9 this->totalNum = 0;
10 }

```

这里介绍以下构造函数的几个参数的含义：minItemNum 表示最小的节点 ID 值，maxItemNum 表示最大的节点 ID 值，maxIteration 表示最大的迭代次数，beta 表示游走系数，blockNum 表示分块的个数，convergence 表示给定的阈值大小，showNum 表示最后希望打印的 TOP 值个数。

接下来读取数据集并得到最大节点、最小节点之类的信息。主要思路是读取文件，然后分别读取节点和出度节点记录其 ID 号，然后通过比较得到整个数据集的最小 ID 和最大 ID。

读取数据集得到最大和最小值

```

1 void PageRank::getItemRange(string dataFile) {
2     ifstream data;
3     data.open(dataFile.c_str(), ios::in);
4     if (!data.is_open())
5     {
6         cerr << "读取文件失败" << endl;
7         exit(0);
8     }
9     int a = INT_MAX, b = -1;
10    int minItem = INT_MAX, maxItem = -1;
11    while (data >> a >> b) {
12        minItem = min(minItem, min(a, b));
13        maxItem = max(maxItem, max(a, b));
14    }
15    this->minItemNum = minItem;
16    this->maxItemNum = maxItem;
17    data.close();
18 }

```


接着需要定义一个将 Item 加入到 matrix 中的函数，这个函数的作用就是将数据集中的数据格式转换为我们定义的 map+set 的矩阵格式。

将 Item 加入 Matrix

```

1 void PageRank::insertItem(int a, int b) {
2     //先检查a,b的值是不是合法
3     if (minItemNum == -1 || maxItemNum == -1) {
4         cerr << "Your minItemNum or maxItemNum must have something wrong!" << endl;
5         exit(0);
6     }
7     else if (a > maxItemNum || a < minItemNum) {
8         cerr << "a form error!" << endl;
9         exit(0);
10    }
11    else if (b > maxItemNum || b < minItemNum) {
12        cerr << "b form error!" << endl;
13        exit(0);
14    }
15    //检查是否存在
16    if (matrix.find(a) != matrix.end()) {
17        matrix[a].insert(b);
18    }
19    else {
20        matrix.insert({ a, set<int>() });
21        matrix[a].insert(b);
22    }
23 }

```

接着是非常重要的 readFile 函数，在这个函数中我们将数据集转为了 matrix 的格式，然后得到初始的 ID 离散列表并将其映射为 ID 紧凑列表并得到其出度数 (如果是 dead points 会将出度数设置为 0)，然后进行数据分块其实就是 matrix 矩阵的分块，通过一次遍历将每个分块的 matrix 分别输出到分块文件中进行存储以便 pagerank 函数进行调用。其中对于映射的思想在上述数据集中以及得到解释，具体可参考4.2。

readFile 函数

```

1 //读取文件
2 void PageRank::readFile(string dataFile) {
3     //如果没有成功得到min和max，或者max<min的话需要先得到min和max
4     if (minItemNum == -1 || maxItemNum == -1 || minItemNum > maxItemNum) {
5         cerr << "Your minItemNum or maxItemNum must have something wrong!" << endl;
6         exit(0);
7     }
8
9     //进行矩阵的读取
10    ifstream data;
11    data.open(dataFile.c_str(), ios::in);
12    int a, b;
13    //用usedItem数组记录使用过的节点

```

```

14  bool* usedItem = new bool[maxItemNum + 2](); //这里使用指针方便释放
15  while (data >> a >> b) {
16      insertItem(a, b);
17      usedItem[a] = true;
18      usedItem[b] = true;
19      totalEdges++;
20  }
21
22  data.close();
23  totalNum = 0;
24  //得到节点数, 并将映射离散化, 即将离散化的节点数分别用1,2,3...表示, 这样有利于节省空间
25  for (int i = minItemNum; i < maxItemNum; i++) {
26      if (usedItem[i]) {
27          nodeToIndex[i] = totalNum; //离散->紧凑
28          indexToNode[totalNum] = i; //紧凑->离散
29          totalNum++;
30      }
31  }
32  delete[] usedItem;
33
34  //进行dead points的判断以及出度数的获得
35  //dead
    points的判断是根据usedItem为true但是matrix中并没有(因为matrix中只记录有出度的)
36  for (int i = 0; i < totalNum; i++) {
37      if (matrix.find(indexToNode[i]) == matrix.end()) {
38          degree.push_back(0);
39      }
40      else {
41          degree.push_back((int)matrix[indexToNode[i]].size());
42      }
43  }
44
45  //进行数据分块(其实就是matrix矩阵的分块, 这样可以优化计算)
46  char fileName[100][300];
47  ofstream blockFile[100]; //设置最多一百个分块
48  for (int i = 0; i < blockNum; i++) {
49      sprintf_s(fileName[i], "block-%d.txt", i);
50      blockFile[i].open(fileName[i]);
51  }
52
53  /**
54  以下是分块示例:
55      1   2   3   4
56      | 1/4 1/2 0   0 |
57      | 1/4 0   1/3 0 |  ==>  | 1/4 1/2 0   0 | + | 1/4 1/2 0   0 |
58      | 1/4 1/2 0   0 |  ==>  | 1/4 0   1/3 0 |   | 1/4 0   1/3 1 |
59      | 1/4 0   1/3 1 |
60  这里能分块的原因是左乘M矩阵, 每行与r相乘, 而存储的时候正是以行进行存储的
61  因此在分块的时候需要记住节点i的所有出度数, 才能得到节点i->节点j的系数, 因此系数等于1/出度数

```

```

62     同时每次计算得到的结果只是 r
        的部分向量，因此每次迭代后都需要重新组合(也可以不用)得到新的 pagerank
63     ***/
64
65     //依次遍历
66     int tmpBlock = 0;
67     for (auto i : matrix) {
68         for (auto j : i.second) {
69             //先将离散化转为紧凑化
70             int index_j = nodeToIndex[j];
71             int index_i = nodeToIndex[i.first];
72             //将index_j化分在对应的块数
73             tmpBlock = index_j / (totalNum / blockNum + 1);
74             blockFile[tmpBlock] << index_i << " " << index_j << endl;
75         }
76     }
77     for (int i = 0; i < blockNum; i++) {
78         blockFile[i].close();
79     }
80     //将node<—>index的映射表打印出来
81     ofstream nodeIndexFile;
82     nodeIndexFile.open("nodeToIndex.txt");
83     for (auto i : nodeToIndex) {
84         nodeIndexFile << i.first << "->" << i.second << endl;
85     }
86     nodeIndexFile.close();
87     cout << "Read Over!" << endl;
88 };

```

做完以上工作以后就可以进行 pagerank 计算了，这里对于 dead ends 和 spider ends 的解决方法在上述原理中已经解释。在 pageRank 函数中，首先是初始化了 pageRank 向量，然后进行迭代(迭代结束的条件是结果小于阈值或者超出迭代次数)，然后根据上述的算法进行迭代计算，这里需要注意的是每次迭代过程中都要对每个 blockFile 文件进行读取(matrix 的一部分)然后进行左乘得到一部分的 pagerank，最后需要对 dead ends 和 spider ends 进行修正。另外每次迭代需要进行归一化的处理这是因为浮点数精度的问题，迭代完成后需要通过映射关系将原始的 ID 映射回来并得到原始节点的 pagerank 值写回 result.txt 文件中。

pagerank 核心函数

```

1 void PageRank::pageRank() {
2     long double alpha = 1 - beta;
3     vector<long double> oldPr; //上一次计算的PageRank
4     oldPr.resize((size_t)totalNum);
5     pr.resize((size_t)totalNum);
6     //初始化pageRank向量
7     for (auto& i : pr) {
8         i = 1.0 / totalNum;
9     }
10 }

```

```

11 long double diff = 1e9; //差值
12 int iterations = 0; //迭代次数
13 int blockSize = totalNum / blockNum + 1;
14 while (diff > convergence && iterations < maxIteration) {
15     long double sumPr = 0.0; //PageRank所有之和
16     for (int i = 0; i < totalNum; i++) {
17         sumPr += pr[i];
18     }
19
20     for (int i = 0; i < totalNum; i++) {
21         oldPr[i] = pr[i] / sumPr;
22     }
23
24     long double addSpider = alpha * 1.0 / totalNum;
25     sumPr = 0.0;
26     diff = 0.0;
27
28
29     ofstream blockFile;
30     for (int block = 0; block < blockNum; block++) {
31         char fileName[300];
32         sprintf_s(fileName, "block-%d.txt", block);
33         blockFile.open(fileName);
34         vector<long double>tmpPr; //部分的PageRank
35         tmpPr.resize((size_t)min(blockSize * (block + 1), totalNum) - blockSize *
36             block + 1);
37         for (auto& i : tmpPr) {
38             i = 0.0;
39         }
40         int minIndex = blockSize * block;
41         int maxIndex = min(blockSize * (block + 1), totalNum);
42
43         int a, b; //index_i, degree, index_j
44
45         while (blockFile >> a >> b) {
46             tmpPr[b - minIndex] += beta * 1.0 / degree[a] * oldPr[a];
47         }
48         //对Dead ends修正
49         for (int i = 0; i < totalNum; i++) {
50             //如果是dead ends
51             if (degree[i] == 0) {
52                 for (int j = minIndex; j < maxIndex; j++) {
53                     tmpPr[j - minIndex] += beta * 1.0 / totalNum * oldPr[i];
54                 }
55             }
56         }
57         //加上dead ends 和 spider ends 的修正
58         for (int i = minIndex; i < maxIndex; i++) {

```

```

59         tmpPr[i - minIndex] += addSpider;
60     }
61
62     for (int i = minIndex; i < maxIndex; i++) {
63         pr[i] = tmpPr[i - minIndex];
64         sumPr += pr[i];
65     }
66     blockFile.close();
67 }
68
69 //进行归一化处理
70 for (int i = 0; i < pr.size(); i++) {
71     pr[i] /= sumPr;
72 }
73
74 for (int i = 0; i < totalNum; i++) {
75     diff += fabs(pr[i] - oldPr[i]);
76 }
77 cout << diff << endl;
78 iterations++;
79 }
80
81 output.resize(totalNum);
82 for (int i = 0; i < totalNum; i++) {
83     output[i] = { -pr[i], i };
84 }
85 sort(output.begin(), output.end());
86 ofstream result;
87 result.open("result.txt");
88 if (showNum < 0 ) {
89     for (int i = 0; i < totalNum; i++) {
90         result << indexToNode[output[i].second] << " " << pr[output[i].second] <<
91             endl;
92         cout << indexToNode[output[i].second] << " " << pr[output[i].second] <<
93             endl;
94     }
95 }
96 else {
97     for (int i = 0; i < totalNum; i++) {
98         result << indexToNode[output[i].second] << " " << pr[output[i].second] <<
99             endl;
100     }
101     for (int i = 0; i < showNum; i++) {
102         cout << indexToNode[output[i].second] << " " << pr[output[i].second] <<
103             endl;
104     }
105 }
106 result.close();

```

```

104     cout << iterations << endl;
105 }

```

以上便是分块 pagerank 算法的过程，其运行结果在下面运行过程中进行给出。

6.2 并行优化

在分块 pagerank 程序运行的过程中，通过 visual studio 自带的分析工具可以发现在分块文件读取的操作上耗费了大量的时间，因此本小组想基于大二学习的并序程序设计的思想进行并行优化。

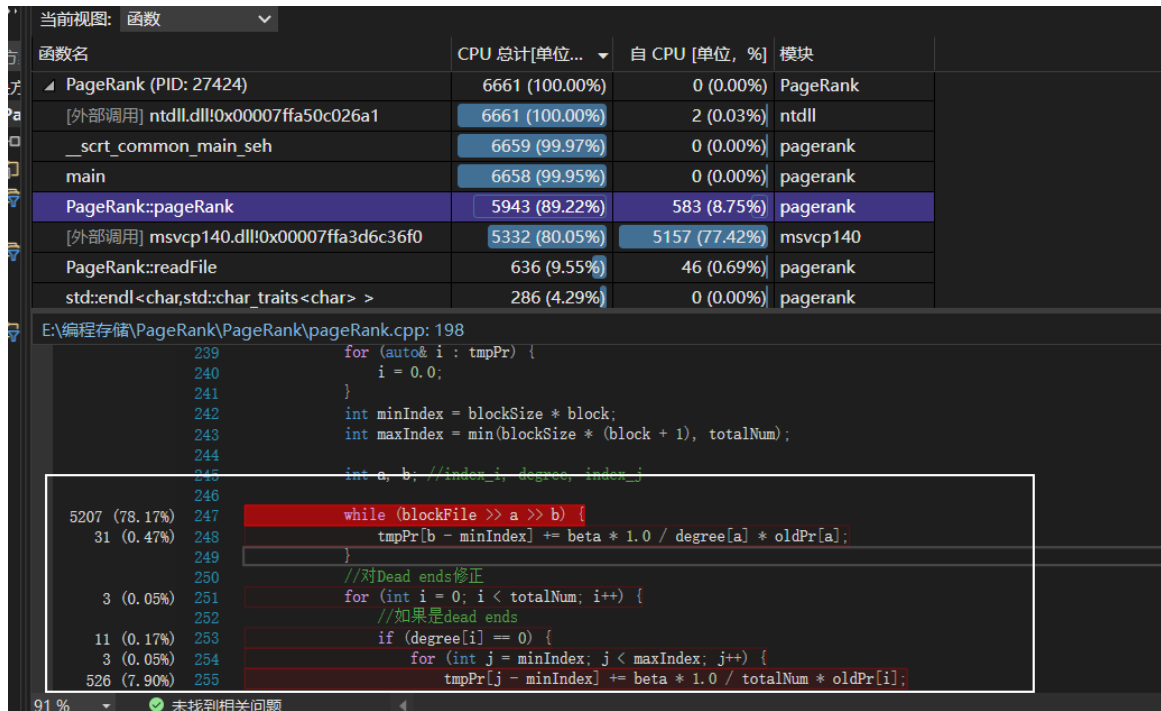


图 6.14: VS 分析主要耗时操作

因此我们尝试将这个操作进行并行化处理，由一个线程去处理几个块的文件，由于每个块的计算是独立的，因此可以去并行处理，但是在最后写回 pagerank 的时候注意要保证原子性的操作，因此需要使用锁去实现。

线程函数

```

1 void PageRank::threadBlock(int blockSize, vector<long double>oldPr, long double
  addSpider, long double& sumPr) {
2     ifstream blockFile;
3     for (int block = 0; block < blockNum; block++) {
4         char fileName[300];
5         sprintf_s(fileName, "block-%d.txt", block);
6         blockFile.open(fileName);
7         vector<long double>tmpPr; // 部分的PageRank
8         tmpPr.resize((size_t)min(blockSize * (block + 1), totalNum) - blockSize *
          block + 1);
9         for (auto& i : tmpPr) {
10             i = 0.0;

```

```

11     }
12     int minIndex = blockSize * block;
13     int maxIndex = min(blockSize * (block + 1), totalNum);
14
15     int a, b; //index_i, degree, index_j
16
17     while (blockFile >> a >> b) {
18         tmpPr[b - minIndex] += beta * 1.0 / degree[a] * oldPr[a];
19     }
20     //对Dead ends修正
21     for (int i = 0; i < totalNum; i++) {
22         //如果是dead ends
23         if (degree[i] == 0) {
24             for (int j = minIndex; j < maxIndex; j++) {
25                 tmpPr[j - minIndex] += beta * 1.0 / totalNum * oldPr[i];
26             }
27         }
28     }
29
30     //加上dead ends 和 spider ends 的修正
31     for (int i = minIndex; i < maxIndex; i++) {
32         tmpPr[i - minIndex] += addSpider;
33     }
34
35     for (int i = minIndex; i < maxIndex; i++) {
36         pr[i] = tmpPr[i - minIndex];
37         sumPr += pr[i];
38     }
39     blockFile.close();
40 }
41 }

```

但是在实际运行过程中发现在当前的数据集下并没有太大的提高，分析原因可能是因为数据集太小，导致由于并行提高的时间没有并行开启线程的时间多，因此导致性能并没有太大的提升。

7 运行过程

7.1 普通算法

因为对 teleport 参数 β 与迭代终止条件的精度 Epsilon 的改变并讨论它们的影响的部分在优化之后进行讨论；所以，不同算法不需要传入参数，teleport 参数 $\beta=0.85$ ，迭代终止条件的精度 Epsilon 设置为 $1e-10$ ，并展示前 100 个结果的 NodeID 以及对应的分数 score，如图7.18所示。

7.2 分块优化算法

在可执行的目录下执行-h 可以得到参数的帮助说明，如图7.15所示

```

PS > .\PageRank\Release> .\PageRank.exe -h
pagerank [-a alpha ] [-c convergence] [-b blocknums] [-m max_iterations] <graph_file>
-a alpha
  default is 0.85
-c convergence
  the convergence criterion default is 1e-5
-m max_iterations
  maximum number of iterations to perform
-b blocknums
  the number of blocknums, for some reason, the number shouldn't be larger than 100
-s show the result
  integer default all
  the output file named out.txt

```

图 7.15: 优化程序参数说明

- -a: 表示游走系数
- -c: 表示阈值大小
- -m: 表示最大迭代次数
- -b: 表示分块个数
- -s: 表示结果在命令行展示 TOP 多少个
- graph_file: 表示数据集文件

因此我们可以通过以下的方式进行执行源程序，如图7.16所示

```

PS > .\PageRank\Release> .\PageRank.exe -a 0.85 -c 1e-10 -m 100 -s 100 Data.txt
Read Over!
Total Nodes => 6262
Total Edges => 83852
MinNodeID => 3 MaxNodeID => 8297
MaxIteration => 100
The iteration is starting...
The 0 iterations => 1.10233
The 1 iterations => 0.351014
The 2 iterations => 0.0953753

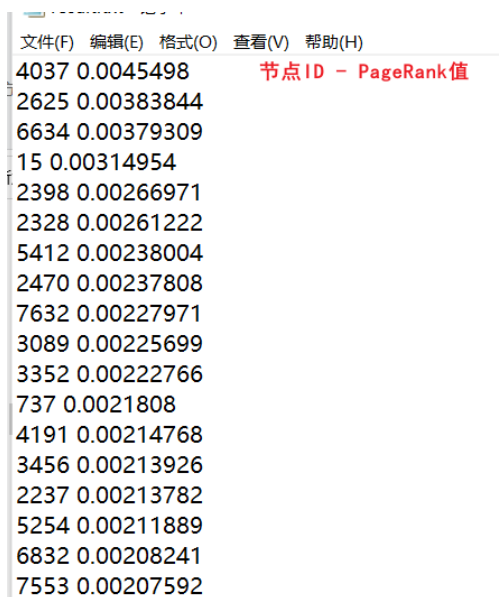
```

图 7.16: 运行程序说明

得到的结果都有说明，因此这里不再赘述。最后在当前文件夹中可以在 block 文件夹中看到很多的 block 文件，这个就是得到的分块文件。同时在可执行程序目录下存在 nodeToIndex.txt 文件夹，这是将离散的节点 ID 映射为紧凑的节点 ID 产生的映射关系数据文件，另外结果文件在 result.txt 中。

名称		修改日期	类型	大小
block	block-file文件	2023/4/27 13:42	文件夹	
Data.txt	Data数据集	2023/3/26 13:48	文本文档	782 KB
nodeToIndex.txt	映射表	2023/4/27 13:42	文本文档	72 KB
PageRank.exe		2023/4/27 13:40	应用程序	54 KB
PageRank.pdb		2023/4/27 13:40	Program Debug ...	1,388 KB
result.txt	结果文件	2023/4/27 13:42	文本文档	109 KB

图 7.17: 运行后目录结构



```

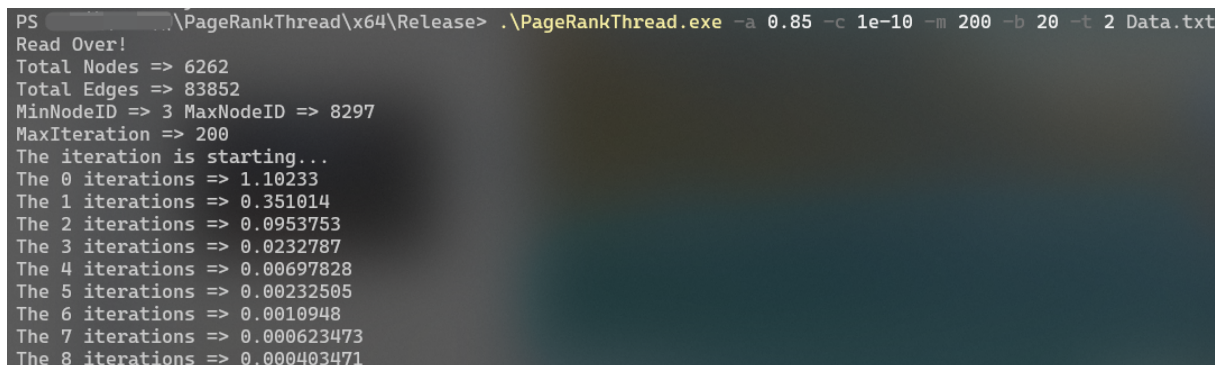
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
4037 0.0045498      节点ID - PageRank值
2625 0.00383844
6634 0.00379309
15 0.00314954
2398 0.00266971
2328 0.00261222
5412 0.00238004
2470 0.00237808
7632 0.00227971
3089 0.00225699
3352 0.00222766
737 0.0021808
4191 0.00214768
3456 0.00213926
2237 0.00213782
5254 0.00211889
6832 0.00208241
7553 0.00207592

```

图 7.18: 得到的结果文件

7.2.1 并行优化算法

这里并行优化算法的可执行程序运行与分块优化算法可执行运行过程一模一样，不同的地方是并行可执行程序多了一个可以设置线程的参数-t，在实际过程中线程数不能设置过高，否则可能会导致数据读写不互斥从而导致得到的差值会越来越大。运行说明如图7.19所示



```

PS C:\Program Files\PageRankThread\x64\Release> .\PageRankThread.exe -a 0.85 -c 1e-10 -m 200 -b 20 -t 2 Data.txt
Read Over!
Total Nodes => 6262
Total Edges => 83852
MinNodeID => 3 MaxNodeID => 8297
MaxIteration => 200
The iteration is starting...
The 0 iterations => 1.10233
The 1 iterations => 0.351014
The 2 iterations => 0.0953753
The 3 iterations => 0.0232787
The 4 iterations => 0.00697828
The 5 iterations => 0.00232505
The 6 iterations => 0.0010948
The 7 iterations => 0.000623473
The 8 iterations => 0.000403471

```

图 7.19: 运行程序说明

8 实验结果分析

在这一小节中我们会分析 teleport 参数 β 、迭代终止条件 ϵ ，分块的个数，以及对于并行优化算法中线程个数分别改变的时候对 PageRank 运行产生的影响。具体的方法为：以默认条件 ($\beta = 0.85, \epsilon = 1 \times 10^{-9}$, 20 个分块个数) 为标准，每次改变三个参数中的一个，而保持另外两个参数不变，来分析程序运行的情况。

8.1 teleport 参数 β 对程序运行结果的影响

保持 $\epsilon = 1 \times 10^{-9}$ ，默认分块个数为 20，使得 β 分别为 0.8, 0.85, 0.9 时，结果分析如下：

运行参数				运行结果		
序号	β	ϵ	块个数	迭代次数	时间消耗	空间消耗
1	0.70	1.00E-9	20	40	3.15582s	8.9375MB
2	0.75	1.00E-9	20	49	3.82841s	8.87891MB
3	0.80	1.00E-9	20	63	4.80761s	8.88281MB
4	0.85	1.00E-9	20	86	6.54899s	8.89062MB
5	0.90	1.00E-9	20	132	9.87683	8.88672MB

可以看出, β 值的变换主要影响了运行的迭代次数和时间开销, 而空间消耗的改变并不明显, 随着 β 值的变大, 程序在计算 PageRank 值时的迭代次数也会相应增加, 时间消耗相应增多。但是对于空间消耗并不会产生很大影响, 且由于块大小只受块个数的影响, 因此不会产生改变。如下图所示: 8.20所示。但是 β 值是是为了解决 spider ends 的问题, 如果设置过小的话很可能导致某一个节点的 pagerank

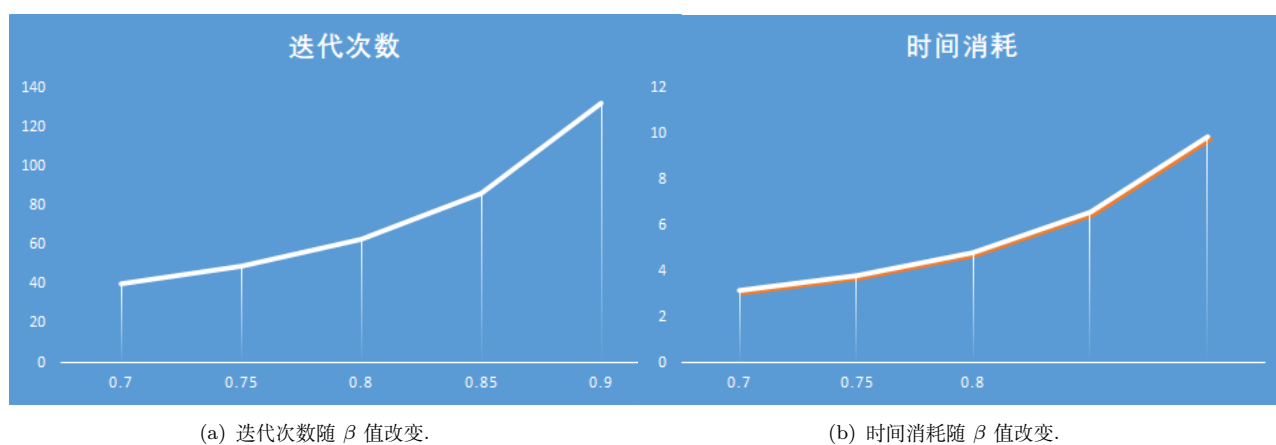


图 8.20: 迭代次数和时间消耗随 β 值改变

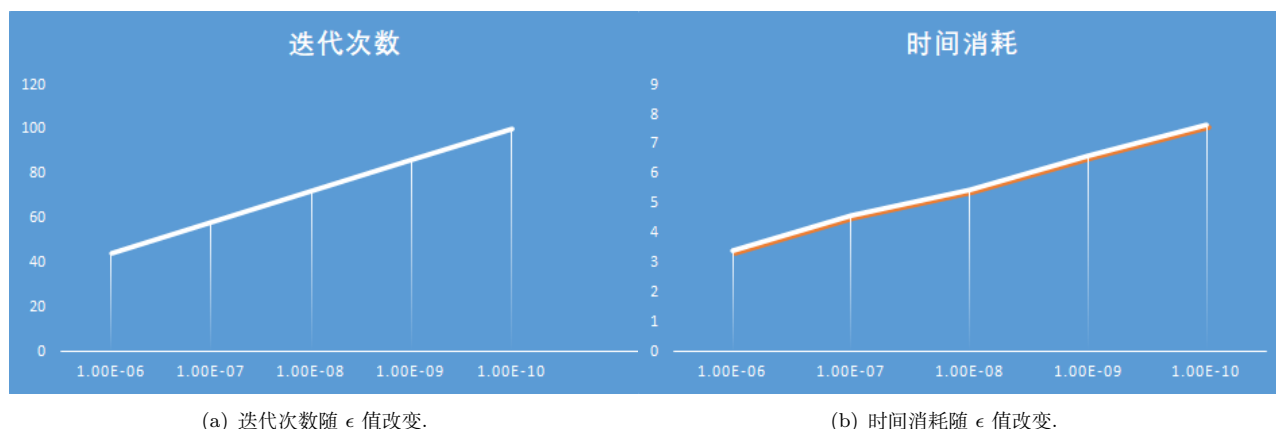
值过于高, 因此为了兼顾两方面的影响, 在实际应用中 β 值的大小一般都设置为 0.85。

8.1.1 迭代终止条件 ϵ 对程序运行结果的影响

保持 $\beta=0.85$, 块的个数保持 20 个不变, 使 ϵ 分别为 1×10^{-6} 1×10^{-7} 1×10^{-8} 1×10^{-9} 1×10^{-10} , 结果分析如下:

运行参数				运行结果		
序号	β	ϵ	块个数	迭代次数	时间消耗	空间消耗
1	0.85	1.00E-6	20	44	3.3838s	8.87891MB
2	0.85	1.00E-7	20	58	4.60398s	8.88672MB
3	0.85	1.00E-8	20	72	5.44543s	8.89453MB
4	0.85	1.00E-9	20	86	6.59246s	8.88672MB
5	0.85	1.00E-10	20	100	7.6414s	8.87109MB

可以看出, ϵ 值的变换主要影响了运行的迭代次数和时间开销, 而空间消耗的改变并不明显, 随着 ϵ 值的变大, 程序在计算 PageRank 值时的迭代次数也会相应增加, 时间消耗相应增多。这是因为随着 ϵ 的减少, 迭代的终止条件越来越严格, 继而导致的是 PageRank 计算的迭代次数成倍的增加, 运行的时间也会相应地增加。但是对于空间消耗并不会产生很大影响, 且由于块大小只受块个数的影响, 因此不会产生改变。如下图所示: 8.21所示。

图 8.21: 迭代次数和时间消耗随 ϵ 值改变

8.2 分块个数对程序运行结果的影响

保持 $\beta = 0.85, \epsilon = 1 \times 10^{-9}$ 不变, 查看不分块、块个数为 10、20、40、50 时候的运行结果, 如下表所示:

运行参数				运行结果		
序号	β	ϵ	块个数	迭代次数	时间消耗	空间消耗
1	0.85	1.00E-9	1	86	6.23311s	8.80859MB
2	0.85	1.00E-9	10	86	6.61961	8.86719MB
3	0.85	1.00E-9	20	86	6.81665s	8.88281MB
4	0.85	1.00E-9	40	86	6.83848s	9.01953MB
5	0.85	1.00E-9	50	86	7.21431s	9.01562MB

与前两个参数 β 和 ϵ 不同, 分块计算的目的是为了将记录矩阵的链表打散、分块, 避免每次计算 PageRank 值过程中产生不必要的磁盘读写, 进而减小内存和时间开销。因此分块操作并不会对节点的 PageRank 值或是迭代次数产生影响, 这一点在上表中已有所印证。此外, 对于块总数来说, 根据程序当中计算的公式: 块总数 = 节点个数/块大小, 可知块总数和块大小之间为反比例函数关系。改变分块个数主要影响体现在内存空间和时间上消耗的变化。首先从时间上来看, 由于不将矩阵分块时, 没有分块的时间消耗和一次迭代时额外读取多次 block 文件的开销, 因此时间开销最小, 而当我们使用分块矩阵算法时, 程序会用一定的时间来做分块工作并且要读取多次 block 文件, 因此导致了时间开销增加; 而随着每个块所能包含的最多节点数逐渐增加, 时间的开销也会越来越小 (主要是读取 block 文件次数变少), 并逐渐逼近不分块情况下的时间开销, 因为不分块也就意味着每个块所能容纳的最大节点数为无穷, 这是符合预期的。在空间消耗方面, 理论上来说使用分块算法所造成的内存空间开销会比不分块时小, 但本次的实验结果与理论相反, 我们通过分析总结出以下两个原因: 一是编译器的优化, 在本次实验中, 编译器的优化选项为 O2 优化, 这会使得程序运行时间和运行内存被优化, 因此这会对程序运行时间和运行内存的分析产生一定影响; 二是本次实验的数据集不够大, 因此没有很好地体现出使用分块算法的优势, 这也就导致了实验结果与预计相反的结果。如下图所示: 8.22所示。

8.3 并行线程对于程序运行结果的影响

在并行程序优化算法程序中, 保持 $\beta = 0.85, \epsilon = 1 \times 10^{-9}$ 以及分块大小为 50 不变, 查看线程数为 1、2、3 的运行结果, 如下表所示:

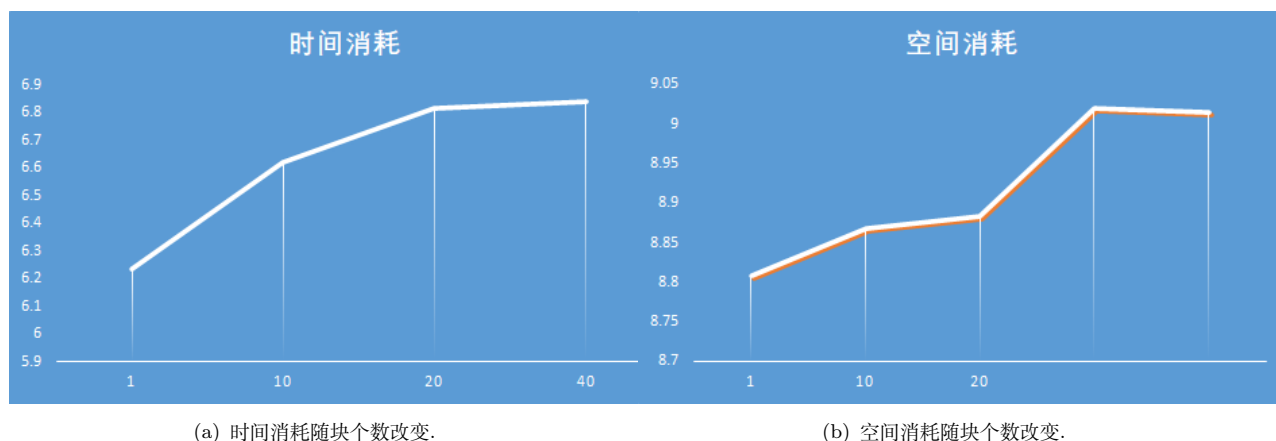


图 8.22: 空间消耗和时间消耗随块个数改变

运行参数					运行结果		
序号	β	ϵ	块个数	线程数	迭代次数	时间消耗	空间消耗
1	0.85	1.00E-9	50	1	86	6.5571s	9.01172MB
2	0.85	1.00E-9	50	2	86	6.36331s	9.67578MB
3	0.85	1.00E-9	50	5	86	7.96487s	9.75MB

可以发现，在线程数为 2 的时候性能有所优化，但当线程数增多的时候反而时间消耗更多了，分析原因主要是因为数据集的大小太小了，因此由线程带来的处理时间减少的部分无法抵消引入线程导致的时间消耗部分。同时由线程的增多空间消耗也会变多，这是因为线程的增加会导致每个线程都会占用一定的内存。而在实际的 Google PageRank 算法运行过程中，Google 主要采用了两种方法来将 PageRank 算法并行化：分布式计算和矩阵分解。这些方法可以利用大量的计算资源，加速计算过程，从而提高计算的时间消耗。在分布式计算中，网页图被分割成多个子图，每个子图由不同的计算节点处理。这种方法可以使计算在多个节点上同时进行，从而减少计算时间。在矩阵分解中，网页图被表示为一个稀疏矩阵，并分解成两个矩阵的乘积。其中一个矩阵表示网页之间的链接关系，另一个矩阵表示每个网页的 PageRank 值。这种方法可以利用矩阵运算的高效性，加速计算过程。这些并行化方法的使用，使得 Google 能够更快地计算出 PageRank 值，从而提高计算的时间消耗。因此本次线程主要是在一个核一个计算节点上进行的，没有做到真正的分布式计算，如果实现分布式并行计算需要用到 Hadoop、Spark 等分布式计算框架实现分布式计算。在 Hadoop 中，可以使用 MapReduce 编程模型，将网页图分割成多个子图，每个子图由不同的计算节点处理。每个节点只需要计算其所处理子图的 PageRank 值，然后将结果合并得到全局的 PageRank 值。在 Spark 中，可以使用 RDD（弹性分布式数据集）进行分布式计算。将网页图表示为一个 RDD，然后使用 Spark 的转换和操作函数对 RDD 进行操作，最后得到 PageRank 值。除了 Hadoop 和 Spark，PageRank 算法还可以使用 MPI 等分布式计算框架实现。在 MPI 中，可以使用消息传递接口进行节点之间的通信和协调，实现分布式计算。具体选择哪种框架和语言实现 PageRank 算法取决于具体的需求和环境，需要根据实际情况进行选择。

9 小组分工

- 杨鑫：实验了分块优化算法和并行分块优化算法的设计和实现，以及对算法原理、结果汇总及分析等绝大部分实验和文档编写工作，以及统筹组内的各项其他工作。

- 吴晨宇：进行了普通 PageRank 算法的设计和实现以及它的原理分析、结果汇总等工作。

10 实验总结

本文将对实现 pagerank 算法实验进行总结。在本次实验中，我们学习了 pagerank 算法的原理和实现方法，并通过编程实现了该算法。通过本次实验，我们收获了以下几点经验和体会。首先，我们深入理解了 pagerank 算法的原理和作用。pagerank 算法是一种用于评估网页重要性的算法，它通过计算网页的链接数量和质量来评估网页的重要性，并根据评估结果进行排序。在实际应用中，pagerank 算法被广泛应用于搜索引擎的排名和推荐系统中。其次，我们掌握了 pagerank 算法的实现方法。在本次实验中，我们使用 C++ 语言编写了 pagerank 算法的代码，并通过对一组网页链接进行模拟计算，验证了算法的正确性。最后，我们总结了本次实验的经验和教训。在实现 pagerank 算法的过程中，我们遇到了许多问题和挑战，例如如何处理大规模数据、如何优化代码性能等。通过不断调试和优化，我们最终成功地实现了 pagerank 算法，并提高了自己的编程能力和解决问题的能力。综上所述，本次实验是一次非常有意义的学习经历，它让我们深入理解了 pagerank 算法的原理和作用，并掌握了该算法的实现方法。我们相信，在今后的学习和工作中，这些经验和体会一定会对我们起到积极的作用。

11 代码仓库

本小组本次实验的所有代码、可执行文件、数据结果、实验报告已经全部上传到小组的仓库中 [Big-DataPageRank](#)。