

大数据计算及应用

作业 1：使用 Wikipedia 数据集计算 PageRank 值

实验报告

May 1, 2020

曹元议 1711425

王雨奇 1711299

段非 1711264

Contents

1	概述	1
1.1	PageRank 算法产生背景	1
1.2	PageRank 算法原理	1
1.3	PageRank 算法的几个问题以及优化	3
1.3.1	Spider Trap 和 Dead End	3
1.3.2	优化稀疏矩阵	4
1.3.3	矩阵的分块计算	5
2	数据集说明	6
3	代码细节说明	7
3.1	分块计算 PageRank	7
3.1.1	Pagerank 类和分块思路介绍	7
3.1.2	Pagerank 类成员函数介绍	9
3.1.3	程序入口	22
3.2	基础 PageRank	23
4	运行过程及云主机截图	27
4.1	参数值默认	27
4.2	只改变参数 β	31
4.2.1	$\beta = 0.8$	31
4.2.2	$\beta = 0.9$	33
4.3	只改变参数 ϵ	35
4.3.1	$\epsilon = 1 \times 10^{-5}$	35
4.3.2	$\epsilon = 1 \times 10^{-7}$	36
4.4	只改变分块大小	38
4.4.1	改变分块大小为 50	38
4.4.2	改变分块大小为 500	40
4.5	运行只优化稀疏矩阵的程序	42
5	实验结果及分析	44
5.1	teleport 参数 β 对程序运行结果的影响	44
5.2	迭代终止条件 ϵ 对程序运行结果的影响	45
5.3	分块的大小对程序运行结果的影响	46
5.4	总结	47

1 概述

1.1 PageRank 算法产生背景

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。

20 世纪 90 年代后期, Larry Page 和 Sergey Brin 提出了 PageRank 算法, 即网页排名、网页级别, 它可以根据网页之间相互的超链接关系来衡量特定网页相对于搜索引擎索引中的其他网页而言的重要程度, Google 用它来体现网页的相关性和重要性, 在搜索引擎优化操作中是经常被用来评估网页优化的成效因素之一。

1.2 PageRank 算法原理

PageRank 算法评价网页质量的方法可以概括为下面内容: 如果一个网页被其他网页链接, 说明该网页的重要性较高; 同时, 被高质量网页链接的网页, 其重要性也会相应提高。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。

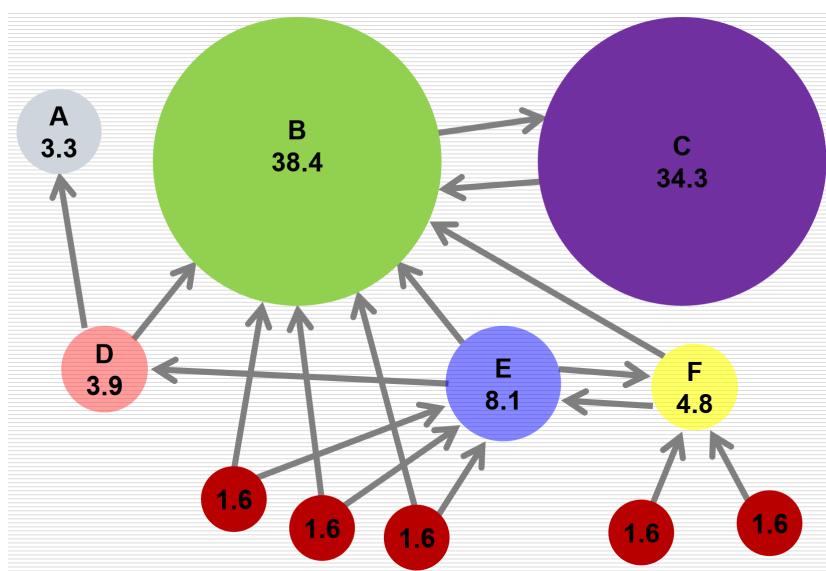


Figure 1: 网页节点重要度示意图

基于如上叙述, 我们可以总结出 PageRank 算法的核心: 某个网页下一时刻的重要度 (r 值) 等于所有链接向它的页面的当前重要度 (r 值) 除以其对应的出链个数的总

和, 即:

$$r_j = \sum_{i \rightarrow j} \frac{r_i}{d_i}$$

其中, d_i 为节点 i 的出链个数。

下图可以更好地解释 PageRank 算法的思想:

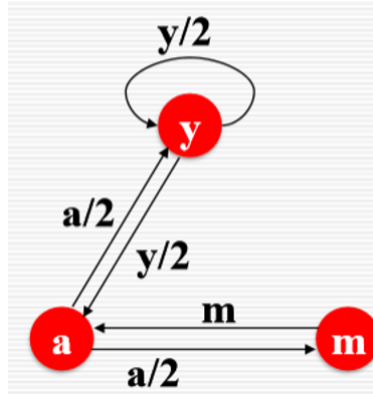


Figure 2: PageRank 算法的思想

从该图中, 我们可以得到三个节点的重要度计算公式:

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

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

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

由上图可知, 每个页面将自己的一部分 r 值传递给其他页面, 我们可以通过计算传递给某个页面的所有 r 值的和来计算出它的 r 值。在 PageRank 算法初始的执行时, 我们可以给每个页面赋予一个初始 r 值—— $\frac{1}{N}$, 其中 N 为页面总数, 然后通过迭代计算得到该页面的 r 值。迭代计算停止的条件为:

新的所有页面的 r 值与旧的所有页面的 r 值之间的差值和小于一个预先设定的值 ϵ , 即:

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

其中 $\|x\|_1 = \sum_{1 \leq i \leq N} |x_i|$ 是 L_1 范式。

在实际计算过程中, 我们可以使用二维矩阵 M 来迭代更新每个页面的 r 值。 M 矩阵的生成方法为: 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 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 $V_n = MV_{n-1}$, 其中 V 是行向量, 记录了每个页面当前的 r 值。

1.3 PageRank 算法的几个问题以及优化

1.3.1 Spider Trap 和 Dead End

Spider Trap 指某个或某几个页面的只有指向自己的出链, 这样会使迭代结果中只有这几个页面的 r 值很高, 其他页面的 r 值为零。

Dead End 就是指一个没有出链的节点, 这时矩阵 M 的该列为零, 这种节点会逐渐吞噬掉整个数据集的能量, 导致迭代的最后结果为零。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, 这种情况下我们的算法就不满足收敛性了。

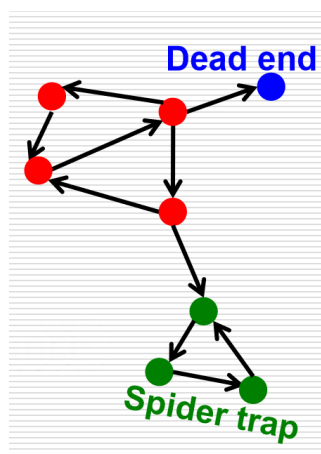


Figure 3: Spider Trap 和 Dead End 示意图

我们可以使用 teleport 来解决这两个问题, 即假设每个页面有很小概率拥有一个指向其他页面的链接, 在每次的计算过程中, 以 β 的概率按照网络的真实情况走, 再以 $(1 - \beta)$ 的概率平均的走向其他所有节点。这样, 对于 Spider Trap 的情况, 可以对每个节点使用 teleport (每迭代一次就为向量 V 的每个分量加上 $\frac{1-\beta}{N}$); 对于 Dead End 的情况, 可以对 Dead End 节点使用 teleport (需要对矩阵 M 全 0 的列进行预处理, 将全 0 列的每个分量改为 $\frac{1}{N}$)。

对于下图 m 节点是 Spider Trap 和 Dead End 的示意图分别如下:

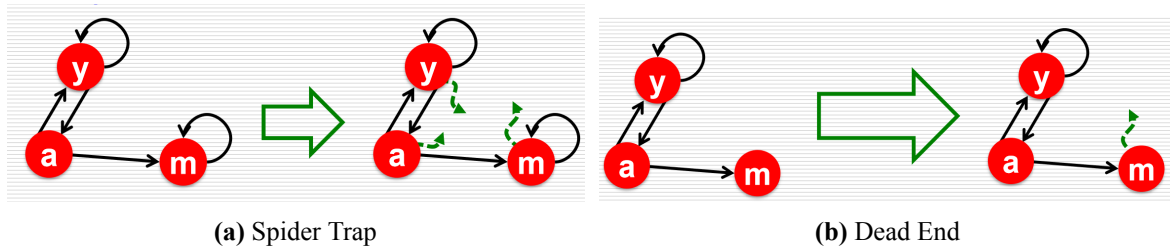


Figure 4: teleport 示意图

事实上，下图的算法可以同时解决这两个问题，并且在有 Dead End 节点的情况下不需要对矩阵 M 进行预处理。即在每次的计算过程中，以 β 的概率按照网络的真实情况走，再将每次因为 Dead End 节点所丢失的 r 值加回给每个节点。如果矩阵 M 当中没有 Dead End 节点，则与对每个节点使用 teleport 的方法等效。本次实验将使用下图的算法。

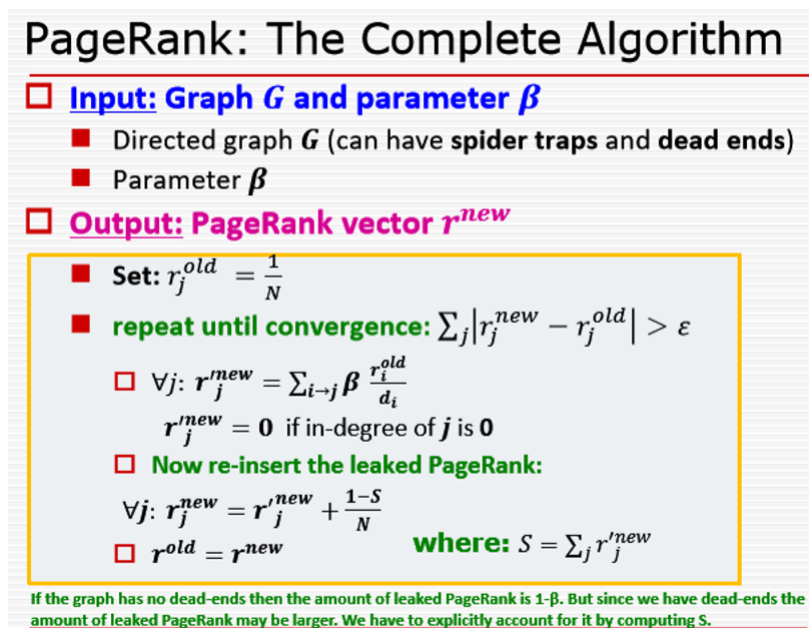


Figure 5: The Complete Algorithm

1.3.2 优化稀疏矩阵

假设数据集中一共包含 N 个节点，那么基于“每个网页平均约有十个外向链接”的理论，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，这足以证明矩阵 M 的稀疏程度。为了减小系统运行时的空间消耗，可以使用链表来存储矩阵 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和示意图如下，本次实验将会

把上面的 The Complete Algorithm 和下面进行一次迭代的算法结合到一起, 此时 r^{new} 的所有分量应初始化为 0。

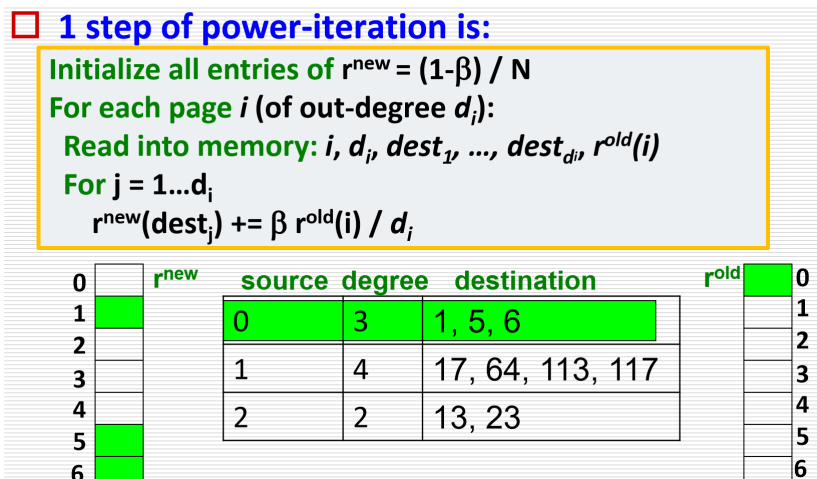


Figure 6: 稀疏矩阵优化示意图

1.3.3 矩阵的分块计算

尽管在上一小节提到我们已经使用链表来优化了稀疏矩阵, 减小了空间损耗, 但对于庞大的原始数据集来说还是不够的。举例来说, 和上一节类似, 假设数据集中一共包含 N 个节点, 那么矩阵 M 的大小是 $10 \times N$, 当数据集中存在 10 亿个网页时, 每个节点的信息至少需要使用 8 个字节存储, 那么其空间消耗就是 $10 \times 10^9 \times 8 = 80GB$ 。

对于如此巨大的数据量, 我们不可能一次性将其全部读入内存进行处理、计算。因此在这种情况下, 选择将矩阵 M 和每次迭代的 r^{old} 存到磁盘中, 这里假设内存可以存下本次迭代的 r^{new} 。但是如果内存小到连本次迭代的 r^{new} 都存不下, 就会选择将 r^{new} 打散, 在这种情况下, r^{new} 被打散成多少块, 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时间复杂度大幅增加。

因此可以采用 Block-Stripe Update Algorithm。该算法可以将矩阵 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 略大)。与只将 r^{new} 打散而不将 M 打散的方式相比, 虽然从磁盘中读取 r^{old} 的次数没变 (这个无法优化), 但是也节省了很大的时间开销。本次实验采用的分块方法也是基于 Block-Stripe Update Algorithm。为了和算法原本的提出背景相契合, 所以计算过程尽量使用文件操作, 即 r^{old} 和分块矩阵中的每一块都存入文件中, 计算时也从 r^{old} 文件和分块矩阵文件中读取相应数据, 每次迭代的 PageRank 值也写入新生成的 r^{old} 文件中。

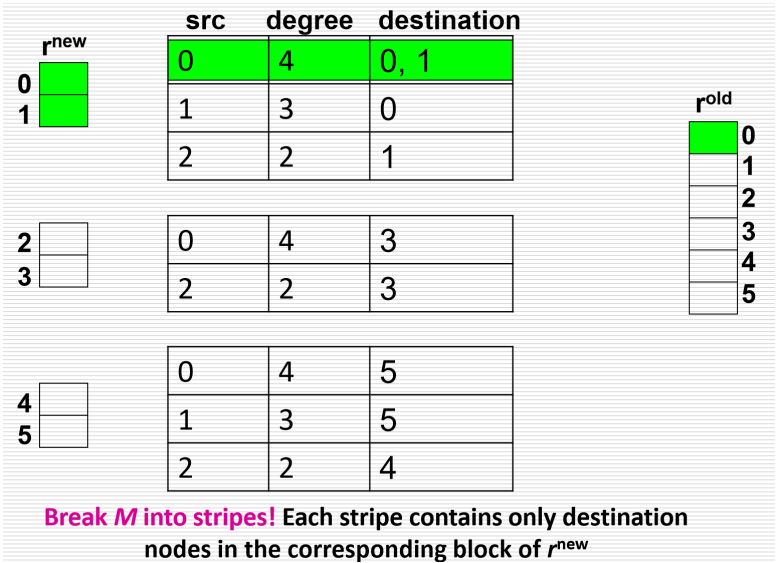


Figure 7: Block-Stripe Update Algorithm 示意图

2 数据集说明

在原始数据集文件WikiData.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。举例来说，下表给出了该数据集的前 8 行数据，在第一行中源节点 30 和目的节点 1412 就表示，该数据集存在一条边是从节点 30 指向节点 1412 的。

源节点	目的节点
30	1412
30	3352
30	5254
30	5543
30	7478
3	28
3	30
3	39

Table 1: 原始数据集文件 WikiData.txt 的前八行数据

经过统计可以了解到，在原始数据集中，出现的最小的节点 ID 为 3，最大的节点 ID 为 8297，在节点 ID 从 0 到 8297 的这些节点中共有 7115 个 ID 被使用（即这些节点

存在出链或入链), 有 6110 个节点存在出链 (因此 Dead End 节点 ID 有 $7115 - 6110 = 1005$ 个)。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的, 由于计算过程中会对节点 ID 重新编号, 这些点将不会被考虑进去, 因此最终生成的结果文件 `result_all.txt` 只有 7115 个结点的信息。详细统计信息在生成的文件 `node_statistics.txt` 中。

3 代码细节说明

本次实验的所有代码使用 C++ 语言编写。

3.1 分块计算 PageRank

以下是分块计算 PageRank 的源码结构:

```
BlockPagerank
├── pagerank.h
├── pagerank.cpp
├── main.cpp
├── mem.h
└── Makefile
```

`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程内存的模块; `BlockPagerank/Makefile` 是在 Linux 系统下编译和运行该程序的规则。

3.1.1 Pagerank 类和分块思路介绍

为了使代码条理更清晰简洁, 我们将 PageRank 算法进行封装, 设计了一个 `Pagerank` 类并附上了相应的类定义。具体情况如下所示:

BlockPagerank/pagerank.h

```
22 class Pagerank
23 {
24 private:
25     /* Pagerank 参数 */
26     double beta; // teleport parameter
27     double epsilon; // 收敛条件系数
28     int maxBlockSize; // 一个分块最多有多少个目的节点
29     int maxIterCount; // 如果源数据导致难以迭代至收敛, 最大的迭代次数
30
31     /* 临时使用变量 */
32     int maxNodeID, minNodeID; // 出现的最大和最小的节点 ID
33     int allNodeCount; // 统计出现的节点 ID 总数
34     int realIterCount; // 实际迭代次数
35     map<int, int> idMap; // 储存实际节点 ID 与节点在程序中的编号的映射
36     int blockCount; // 实际分块个数
37     bool special; // 判断 allNodeCount 能不能整除 maxBlockSize
38 public:
39     Pagerank();
40     ~Pagerank();
41     void setBeta(double beta);
42     void setEpsilon(double epsilon);
43     void setMaxBlockSize(double maxBlockSize);
44     void setMaxIterCount(double maxIterCount);
45     void loadMatrixFromFile(char *filename); // 从文件中读取矩阵信息
46     void calculate(); // 计算 Pagerank 值
47     void writeResultIntoFile(); // 将计算结果写入到文件
48 };
```

其中, 类名为Pagerank, 双精度值beta用来代表我们的 teleport 参数。而双精度值epsilon用来表示进行迭代运算后需要达到的精度要求。而变量maxBlockSize用来存储一个分块最多有多少个目的节点, maxIterCount用来表示最大的迭代次数(这样, 在源数据难以迭代至收敛的情况下, 程序也可以自动退出迭代, 从而退出程序)。

maxNodeID和minNodeID则用来表示出现的最大和最小的节点 ID, allNodeCount表示出现的节点的总数, 而realIterCount则表示实际迭代次数。而映射idMap用来存储实际节点 ID 与节点在程序中的编号的映射, blockCount表示实际分块的个数。为了保证maxBlockSize和blockCount和allNodeCount之间的存储关系, 一个blockCount需要满足里面足够的maxBlockSize的节点大小, 因此引入bool变量special来判断allNodeCount能不能整除maxBlockSize。上述就是全部的Pagerank类的成员变量。

在此之外,Pagerank类成员函数主要有setBeta()、setEpsilon()、setMaxBlockSize()、setMaxIterCount(),这四个是设置相应的类变量参数。loadMatrixFromFile()则是从文件中读取矩阵信息,calculate()用来计算 PageRank 值,而writeResultIntoFile()则是将计算结果写入到文件。

简要介绍一下分块思路:如果allNodeCount能整除maxBlockSize,则设置special变量的值为false,分块个数blockCount的值为allNodeCount / maxBlockSize,此时每个块的目的节点 ID 个数为maxBlockCount;反之设置special变量的值为true,分块个数是allNodeCount / maxBlockSize + 1,此时块0到blockCount - 2的目的节点 ID 个数为maxBlockCount,最后一块(块号为blockCount - 1)的目的节点 ID 个数为allNodeCount % maxBlockCount。所有分块的信息将保存在文件夹BlockPagerank/blocks中,其中第 0 个分块对应的文件为block0.txt,以此类推。在这样的方法之下,一个目的节点在所属块的 ID 为该节点在程序中的 ID 模除所属块的块号。需要注意的是,在分块过程当中和计算的时候,程序会按照源文件节点 ID 大小的相对顺序重新编排 ID,例如在源文件WikiData.txt中出现的最小节点 ID 为 3,则这个节点在程序中的 ID 为 0;如果源文件当中的节点 ID 为 4,则这个节点在程序中的 ID 为 1,以此类推,这个映射关系保存在idMap当中。

3.1.2 Pagerank 类成员函数介绍

在这一部分,我们针对Pagerank类进行更细致的讲解分析,

首先,Pagerank类的构造函数如下,用于初始化 PageRank 参数和创建生成的分块文件夹和rold文件夹:

```
BlockPagerank/pagerank.cpp
19 Pagerank::Pagerank()
20 {
21     this->beta = 0.85;
22     this->epsilon = 1e-9;
23     this->maxBlockSize = 100;
24     this->maxIterCount = 1000;
25 #ifdef _WIN32
26     if (_access("blocks", 0) == -1)
27         _mkdir("blocks");
28     else
29         system("del blocks /Q");
30     if (_access("rold", 0) == -1)
31         _mkdir("rold");
32     else
```

```
33     system("del rold /Q");
34 #else
35     if (access("blocks", 0) == -1)
36         system("mkdir -p blocks");
37     else
38         system("rm -rf blocks/*");
39     if (access("rold", 0) == -1)
40         system("mkdir -p rold");
41     else
42         system("rm -rf rold/*");
43 #endif
44 }
```

其中`beta`值为0.85,这是常规的 PageRank 算法推荐的参数值。`epsilon`设置为`1e-9`,`maxBlockSize`设置为 100,表示一个分块最多有 100 个目的节点。`maxIterCount`设置为 1000。在这个基础上,我们针对 Linux 和 Windows 分别设置了相互独立地设置创建分块文件夹和`rold`文件夹的命令,提高程序的通用性。

接下来是析构函数和`setBeta()`、`setEpsilon()`、`setMaxBlockSize()`、`setMaxIterCount()`。其中析构函数用于清除映射`idMap`的内容,后面四个函数的定义是设置四个核心参数值。

BlockPagerank/pagerank.cpp

```
46 Pagerank::~Pagerank()
47 {
48     idMap.clear();
49 }
50
51 void Pagerank::setBeta(double beta)
52 {
53     this->beta = beta;
54 }
55
56 void Pagerank::setEpsilon(double epsilon)
57 {
58     this->epsilon = epsilon;
59 }
60 void Pagerank::setMaxBlockSize(double maxBlockSize)
61 {
```

```

62     this->maxBlockSize = maxBlockSize;
63 }
64
65 void Pagerank::setMaxIterCount(double maxIterCount)
66 {
67     this->maxIterCount = maxIterCount;
68 }

```

接下来是函数loadMatrixFromFile(), PageRank 算法读入原始数据, 并进行节点统计的分析, 进行相关前置计算, 并且将数据事先进行分块。

BlockPagerank/pagerank.cpp

```

70 void Pagerank::loadMatrixFromFile(char *filename)
71 {
72     assert(filename);
73     ifstream infile(filename);
74     if (!infile.is_open())
75     {
76         perror(filename);
77         exit(1);
78     }
79     cout << "Loading matrix from file " << filename << ".....";
80 #ifdef TIME_TEST
81     start_time = clock();
82 #endif // TIME_TEST

```

程序开始是一个打开文件函数, 这里加入了if (!infile.is_open())的容错机制, 如果未能找到源文件, 则输出错误信息。然后打开时间函数, 开始计时。

BlockPagerank/pagerank.cpp

```

83     map<int, set<int>> allEdges;
84     int fromNodeID, toNodeID;
85     maxNodeID = 0;
86     minNodeID = INT_MAX;
87     allNodeCount = 0;
88     bool idStat[10000] = { 0 };

```

接下来设置allEdges映射结构暂存从文件中读入的数据, 运用节点值和该节点的出度节点的值的集合分别作为key和value存入allEdges。将maxNodeID设置为 0,

minNodeID 设置为 INT_MAX (最大值)。

BlockPagerank/pagerank.cpp

```
89     while (infile >> fromNodeID >> toNodeID)
90     {
91         allEdges[fromNodeID].insert(toNodeID);
92         idStat[fromNodeID] = true;
93         idStat[toNodeID] = true;
94         if (max(maxNodeID, fromNodeID) == fromNodeID)
95         {
96             maxNodeID = fromNodeID;
97         }
98         if (max(maxNodeID, toNodeID) == toNodeID)
99         {
100             maxNodeID = toNodeID;
101         }
102         if (min(minNodeID, fromNodeID) == fromNodeID)
103         {
104             minNodeID = fromNodeID;
105         }
106         if (min(minNodeID, toNodeID) == toNodeID)
107         {
108             minNodeID = toNodeID;
109         }
110     }
111     infile.close();
```

读入原始数据, 并存入相关数据结构即可, 顺便计算出每个节点的出度和入度, 最大节点数和最小节点数。

BlockPagerank/pagerank.cpp

```
114     for (int i = 0; i < 10000; i++)
115     {
116         if (idStat[i])
117         {
118             idMap[i] = allNodeCount;
119             allNodeCount++;
120         }
121     }
122     #ifdef TIME_TEST
123     end_time = clock();
```

```

124     time_ = (double)(end_time - start_time) / CLOCKS_PER_SEC;
125     total_time += time_;
126     cout << "done." << endl;
127     cout << "Time cost in loading matrix: " << time_ << "s" << endl;
128 #else
129     cout << "done." << endl;
130 #endif // TIME_TEST

```

接下来按照节点 ID 的大小顺序为每个节点重新编号，建立节点 ID 和序号之间的映射关系，并且计时器会得出上述过程的具体耗时。为实验进行参考。

BlockPagerank/pagerank.cpp

```

133     ofstream outfile;
134 #ifdef STAT
135     cout << "Generating node statistics file.....";
136 #ifdef TIME_TEST
137     start_time = clock();
138 #endif // TIME_TEST
139     outfile.open("node_statistics.txt");
140     outfile << "Source node count: " << allEdges.size() << endl;
141     outfile << "All node count: " << allNodeCount << endl;
142     outfile << "Max node ID: " << maxNodeID << endl;
143     outfile << "Min node ID: " << minNodeID << endl;
144     for (int i = 0; i < 10000; i++)
145     {
146         outfile << "[ id: " << i << " ";
147         if (!idStat[i])
148         {
149             outfile << "not used";
150         }
151         else
152         {
153             outfile << "used" << " ";
154             if (allEdges.find(i) == allEdges.end())
155             {
156                 outfile << "degree: 0";
157             }
158             else
159             {

```

```

160         outfile << "degree: " << allEdges[i].size() << " outNode: ";
161         set<int>::iterator iter = allEdges[i].begin();
162         while (iter != allEdges[i].end())
163         {
164             outfile << *iter << ", ";
165             iter++;
166         }
167     }
168 }
169 outfile << " ]" << endl;
170 }
171 outfile.close();

```

按照一定格式输出节点统计信息即可, 统计信息包含节点的出度节点和入度节点情况, 最大节点值和最小节点值等信息, 详见生成的node_statistics.txt中。

BlockPagerank/pagerank.cpp

```

183     cout << "Blocking matrix.....";
184 #ifdef TIME_TEST
185     start_time = clock();
186 #endif // TIME_TEST
187     blockCount = allNodeCount / maxBlockSize; // 计算分块个数
188     special = allNodeCount % maxBlockSize != 0; // 判断能不能整除
189     if (special) // 校正
190     {
191         blockCount++;
192     }
193     vector<ofstream> makeBlocks(blockCount);
194     for (int i = 0; i < blockCount; i++) // 提前打开分块文件, 避免 I/O 过于密集, 减少分块时间
195     {
196         char tempstr[50] = { 0 };
197         sprintf(tempstr, "blocks/block%d.txt", i);
198         makeBlocks[i].open(tempstr, ios::app);
199     }
200     map<int, set<int>>::iterator iter1 = allEdges.begin();
201     map<int, set<int>> edgesInBlock;

```

接下来开始对处理后数据进行分块处理, 确定分块个数, 并打开所有分块文件blocks/block%d.txt。

BlockPagerank/pagerank.cpp

```

202 while (iter1 != allEdges.end()) // 遍历 allEdges 的每个条目
203 {
204     // 提取出 allEdges 该源节点条目所在的块信息, 注意这里使用的节点序号是上面
    // 重新编的序号
205     set<int>::iterator iter2 = iter1->second.begin();
206     while (iter2 != iter1->second.end())
207     {
208         int temp = idMap[*iter2];
209         edgesInBlock[temp / maxBlockSize].insert(temp);
210         iter2++;
211     }
212
213     // 将分块信息写入到文件
214     map<int, set<int>>::iterator iter3 = edgesInBlock.begin();
215     while (iter3 != edgesInBlock.end())
216     {
217         int blockID = iter3->first;
218         // 分块文件每一行的格式为:
219         // 源节点 ID 源节点出度 源节点在这个块的出度数 目的节点 1ID 目的
        // 节点 2ID .....
220         makeBlocks[blockID] << idMap[iter1->first] << " " <<
            iter1->second.size() << " ";
221         makeBlocks[blockID] << iter3->second.size() << " ";
222         set<int>::iterator iter4 = iter3->second.begin();
223         while (iter4 != iter3->second.end())
224         {
225             makeBlocks[blockID] << *iter4 << " ";
226             iter4++;
227         }
228         makeBlocks[blockID] << endl;
229         iter3++;
230     }
231     edgesInBlock.clear();
232     iter1++;
233 }
234 allEdges.clear();
235 edgesInBlock.clear();
236 for (int i = 0; i < blockCount; i++)
237 {

```

```

238         makeBlocks[i].close();
239     }
240     makeBlocks.clear();
241     #ifdef TIME_TEST
242         end_time = clock();
243         time_ = (double)(end_time - start_time) / CLOCKS_PER_SEC;
244         total_time += time_;
245         cout << "done." << endl;
246         cout << "Time cost in blocking matrix: " << time_ << "s" << endl;
247     #else
248         cout << "done." << endl;
249     #endif // TIME_TEST
250 }

```

后续将生成分块文件，遍历allEdges的每个条目，提取出allEdges中该源节点条目所在的块信息到edgesInBlock中，按照新的编号进行计算，并按照格式向相应的分块文件输出每个节点具体所在的数据块信息。分块文件每一行的格式为：

源节点ID 源节点出度 源节点在这个块的出度数 目的节点1ID 目的节点2ID

接下来是calculate()函数的计算细节：

BlockPagerank/pagerank.cpp

```

252 void Pagerank::calculate()
253 {
254     cout << "Calculating pagerank....." << endl;
255     #ifdef TIME_TEST
256         start_time = clock();
257     #endif // TIME_TEST
258     /* 初始化变量 */
259     realIterCount = 0; // 迭代次数
260     double dvalue = 0.0; // convergence
261     const char *curr_rolD = "rolD/rolD%d.dat"; // 每次迭代的 rolD 都从单独的文件中存取，由句柄 rolD 和 rnew 进行操作
262     const char *currBlock = "blocks/block%d.txt"; // 分块文件
263     char tempstr[50] = { 0 }; // 字符串缓冲区
264     vector<double> rnew_; // 计算过程中，单独的分块对应的 rnew
265     double init = 1.0 / (double)allNodeCount;
266     fstream rolD; // rolD 对应的句柄
267     fstream rnew; // 本次迭代的 rnew 对应的句柄
268     ifstream readBlock; // 读取分块矩阵的句柄

```

```

269
270 // 初始化向量 rold 和 rnew
271 sprintf(tempstr, curr_rold, realIterCount);
272 rold.open(tempstr, ios::in | ios::out | ios::binary | ios::trunc);
273 sprintf(tempstr, curr_rold, realIterCount + 1);
274 rnew.open(tempstr, ios::in | ios::out | ios::binary | ios::trunc);
275 rold.seekp(0, ios::beg);
276 rnew.seekp(0, ios::beg);
277 rold.seekg(0, ios::beg);
278 rnew.seekg(0, ios::beg);
279 for (int i = 0; i < allNodeCount; i++)
280 {
281     rold.write((char*)&init, sizeof(double));
282 }
283
284 while (realIterCount < maxIterCount)
285 {
286     cout << "Iteration: " << realIterCount << endl;
287     double leaked = 0.0; // 用于计算 Leaked pagerank
288     double temprank = 0.0;

```

首先,做好将分块数据读入内存的准备,初始化迭代次数、每次迭代的rold都从单独的文件rold/rold%d.dat中存取,文件由句柄rold和rnew进行操作,以及初始化分块文件路径,设置好文件指针,将每个节点初始值写入第一个rold文件rold/rold0.dat中,打开本次迭代新生成的rold文件。

BlockPagerank/pagerank.cpp

```

289 #ifdef DEBUG
290     cout << "Calculating blocks....." << endl;
291 #endif // DEBUG
292 for (int i = 0; i < blockCount; i++) // 分块计算
293 {
294     sprintf(tempstr, currBlock, i);
295     readBlock.open(tempstr);
296     int src = 0, degree = 0, size = 0;
297     int rnew_size = special && i == blockCount - 1 ?
        allNodeCount%maxBlockSize : maxBlockSize;
298     rnew_.resize(rnew_size, 0.0); // 当前块的 rnew 初始化为 0
299     while (readBlock >> src >> degree >> size)

```

```

300     {
301         int dst = 0;
302         rold.seekg(src*sizeof(double), ios::beg);
303         rold.read((char*)&temprank, sizeof(double)); // 从文件中读入
304         rold 值
305         while (size)
306         {
307             readBlock >> dst;
308             rnew_[dst % maxBlockSize] += beta*temprank /
309             (double)degree;
310             size--;
311         }
312     }
313     for (int j = 0; j < rnew_.size(); j++)
314     {
315         leaked += rnew_[j];
316         rnew.write((char*)&rnew_[j], sizeof(double));
317     }
318     rnew_.clear();
319     readBlock.close();
320 }

```

接下来就是按照分块个数`blockCount`，依次计算每个分块中节点的 PageRank 值。对于其中一个块，先将该块从文件`blocks/block%d.txt`中读入内存（具体是数据块中每个节点和该节点的出度节点和入度节点），以及从上次迭代的`rold`文件将源节点的 PageRank 值读入内存，对相应的节点按照 PageRank 基础公式进行更新计算并存入当前块的`rnew_`。然后对该分块的每一个节点的加和加在`leaked`变量上，将该分块的 PageRank 值从`rnew_`写入到本次迭代新生成的`rold`文件，为修正 PageRank 值做准备（依次将因 Dead End 流失的 PageRank 值加回来）。以下通过`leaked`修正 PageRank 值（通过同时读写新生成的`rold`文件）：

BlockPagerank/pagerank.cpp

```

320     // 所有分块计算完之后将 Leaked pagerank 加回来
321     #ifdef DEBUG
322         cout << "Calculating leaked pagerank....." << endl;
323     #endif // DEBUG
324     leaked = (1.0 - leaked) / (double)allNodeCount;
325     rnew.seekg(0, ios::beg);

```

```
326     rnew.seekp(0, ios::beg);
327     for (int j = 0; j < allNodeCount; j++)
328     {
329         double curr = 0;
330         rnew.seekg(j*sizeof(double), ios::beg);
331         rnew.read((char*)&curr, sizeof(double));
332         curr += leaked;
333         rnew.seekp(j*sizeof(double), ios::beg);
334         rnew.write((char*)&curr, sizeof(double));
335     }
336
337     // 计算 convergence
338     #ifdef DEBUG
339         cout << "Calculating convergence....." << endl;
340     #endif // DEBUG
341     rnew.seekg(0, ios::beg);
342     rold.seekg(0, ios::beg);
343     dvalue = 0.0;
344     while (true)
345     {
346         double old, new_;
347         rnew.read((char*)&new_, sizeof(double));
348         rold.read((char*)&old, sizeof(double));
349         if (rnew.eof() || rold.eof())
350             break;
351         dvalue += fabs(new_ - old);
352     }
353     rnew.close();
354     rold.close();
355     realIterCount++;
356
357     // 判断是否收敛
358     cout << "Convergence: " << dvalue << endl;
359     if (dvalue <= epsilon)
360         break;
361
362     // 重置 rold 和 rnew
363     sprintf(tempstr, curr_rold, realIterCount);
364     rold.open(tempstr, ios::in | ios::binary);
365     sprintf(tempstr, curr_rold, realIterCount + 1);
```

```
366     rnew.open(tempstr, ios::in | ios::out | ios::binary | ios::trunc);
367     rnew.seekp(0, ios::beg);
368     rnew.seekg(0, ios::beg);
369     rold.seekg(0, ios::beg);
370 }
371 if (rold.is_open())
372     rold.close();
373 if (rnew.is_open())
374     rnew.close();
375 #ifdef TIME_TEST
376     end_time = clock();
377     time_ = (double)(end_time - start_time) / CLOCKS_PER_SEC;
378     total_time += time_;
379     cout << "Calculate pagerank complete." << endl;
380     cout << "Time cost in calculate pagerank: " << time_ << "s" << endl;
381 #else
382     cout << "Calculate pagerank complete." << endl;
383 #endif // TIME_TEST
384     cout << "Real iteration count: " << realIterCount << endl;
385 #ifdef MEM_TEST
386     double currentSize = (double)getCurrentRSS() / 1024 / 1024;
387     cout << "Calculate memory cost: " << currentSize << " MB" << endl;
388 #endif // MEM_TEST
389 }
```

最后就是判断该次计算的数据是否收敛, 如果不收敛, 则打开新生成的`rold`文件作为计算时要读取的`rold`向量和下次迭代新生成的`rold`文件, 为下一次计算做准备。如果收敛则提前退出最外层循环。至此, 所有节点的 PageRank 值计算结束, 然后输出迭代次数, 测量当前使用物理内存的总量和计算需要的时间。

后续的`writeResultIntoFile()`函数为生成结果并输出: 先将最后生成的`rold`文件(对应`realIterCount`)全部读入内存, 并将程序中使用的节点 ID 还原回源数据中的节点 ID, 按照 PageRank 值从大到小对数据进行排序, 然后按照一定的格式将排序后的结果输出到`result_all.txt`和`result_top100.txt`中。`result_top100.txt`的内容就是`result_all.txt`的前 100 行。最后统计所有节点 PageRank 的总和, 用于结果分析。

BlockPagerank/pagerank.cpp

```
391 void Pagerank::writeResultIntoFile()
392 {
393     ofstream outfile1("result_all.txt");
394     ofstream outfile2("result_top100.txt");
395     cout << "Writing result into file result_all.txt and
        result_top100.txt.....";
396 #ifdef TIME_TEST
397     start_time = clock();
398 #endif // TIME_TEST
399     char tempstr[50] = { 0 };
400     vector<pair<double, int>> result(allNodeCount, pair<double, int>(0.0,
        0));
401     sprintf(tempstr, "rold/rold%d.dat", realIterCount);
402     ifstream readRank(tempstr, ios::in | ios::binary); // 从最后一个 rold 导
        入 Pagerank 值
403     double temp;
404     readRank.seekg(0, ios::beg);
405     int i = 0;
406     while (true)
407     {
408         readRank.read((char*)&temp, sizeof(double));
409         if (readRank.eof())
410             break;
411         result[i].first = temp;
412         // 根据重新编的号查找实际节点 ID
413         map<int, int>::iterator idMap_inv = find_if(idMap.begin(),
            idMap.end(),
414             [i](const map<int, int>::value_type item)
415             {
416                 return item.second == i;
417             });
418         if (idMap_inv != idMap.end())
419         {
420             result[i].second = (*idMap_inv).first;
421         }
422         i++;
423     }
424     readRank.close();
425 }
```

```
426     sort(result.rbegin(), result.rend()); // 对结果由大到小进行排序, sort 默认
427     对 first 进行排序
428
429     // 输出结果
430     // 格式为: [NodeID]    [Score]
431     double sum = 0.0;
432     int count = 0;
433     for (int i = 0; i < result.size(); i++)
434     {
435         outfile1 << "[" << result[i].second << "]"    [" << result[i].first <<
436         "]" << endl;
437         if (count < 100)
438             outfile2 << "[" << result[i].second << "]"    [" <<
439             result[i].first << "]" << endl;
440         sum += result[i].first;
441         count++;
442     }
443     outfile1.close();
444     outfile2.close();
445     result.clear();
446 #ifdef TIME_TEST
447     end_time = clock();
448     time_ = (double)(end_time - start_time) / CLOCKS_PER_SEC;
449     total_time += time_;
450     cout << "done." << endl;
451     cout << "The sum of pagerank: " << sum << endl;
452     cout << "Time cost in writing result: " << time_ << "s" << endl;
453 #else
454     cout << "done." << endl;
455     cout << "The sum of pagerank: " << sum << endl;
456 #endif // TIME_TEST
457 }
```

3.1.3 程序入口

以下是程序入口函数`main()`函数。首先通过`parse_args()`解析命令行参数并设置程序运行的参数值和输入文件（如果不指定参数则使用默认参数，该函数由于和实验原理相关度不大因此省略，具体可以查询源代码文件），然后从文件中读取矩阵信息、计算 PageRank 值、输出结果到文件。最终统计程序运行的总时间和程序占用的峰值内存

大小（也就是程序运行时最多使用了多少物理内存）。

在BlockPagerank/main.cpp的第 103 行即是源数据文件的调用位置！

```
BlockPagerank/main.cpp
100 int main(int argc, char *argv[])
101 {
102     parse_args(argc, argv);
103     p.loadMatrixFromFile(srcFile);
104     p.calculate();
105     p.writeResultIntoFile();
106 #ifdef TIME_TEST
107     cout << "Total time cost: " << total_time << "s" << endl;
108 #endif // TIME_TEST
109 #ifdef MEM_TEST
110     double peakSize = (double)getPeakRSS() / 1024 / 1024;
111     cout << "Total memory cost: " << peakSize << " MB" << endl;
112 #endif // MEM_TEST
113     return 0;
114 }
```

3.2 基础 PageRank

以下是基础 PageRank 的源码结构，和分块计算 PageRank 的源码结构大致相同：

```
BasicPagerank
├── pagerank.h
├── pagerank.cpp
├── main.cpp
├── mem.h
└── Makefile
```

为了从运行速度，内存占用等代码运行性能方面对比我们的分块 PageRank 算法，我们又实现了一遍基础 PageRank 算法。基础 PageRank 算法与分块 PageRank 算法代码结构基本相同，主要区别体现在loadMatrixFromFile()和calculate()函数上面，并且类定义中没有与分块相关的变量。其中，对于基础 PageRank 算法的loadMatrixFromFile()函数部分，我们不需要对处理的源数据进行分块存储，而是直接生成稀疏矩阵文件matrix

.txt, 这样是与分块 PageRank 算法互相区别, 因而loadMatrixFromFile()函数在基础 PageRank 算法中的实现要比分块 PageRank 算法简单, 其核心代码如下所示:

```
BasicPagerank/pagerank.cpp

82     for (int i = 0; i < 10000; i++)
83     {
84         if (idStat[i])
85         {
86             idMap[i] = allNodeCount;
87             allNodeCount++;
88         }
89     }
90
91     ofstream outfile;
92     map<int, set<int>>::iterator iter1 = allEdges.begin();
93     outfile.open("matrix.txt");
94     while (iter1 != allEdges.end()) // 遍历 allEdges 的每个条目
95     {
96         // 将矩阵信息写入到文件中, 注意这里使用的节点序号是上面重新编的序号
97         outfile << idMap[iter1->first] << " " << iter1->second.size() << "
98         ";
99         set<int>::iterator iter2 = iter1->second.begin();
100        while (iter2 != iter1->second.end())
101        {
102            outfile << idMap[*iter2] << " ";
103            iter2++;
104        }
105        outfile << endl;
106        iter1++;
107    }
108    outfile.close();
```

上述代码即是将数据信息存入内存中, 没有进行分块处理, 处理后生成的稀疏矩阵保存在matrix.txt中。后续进行运算时直接将该文件的数据全部读入内存即可。

第二个区别是在calculate()运算阶段:

```
BasicPagerank/pagerank.cpp

177     realIterCount = 0; // 迭代次数
178     double dvalue = 0.0; // convergence
179     vector<double> rnew; // 计算过程中对应的 rnew
```

```

180     double init = 1.0 / (double)allNodeCount;
181     fstream rold; // rold 对应的句柄
182     ifstream readMatrix; // 读取稀疏矩阵的句柄

```

首先对变量进行初始化, 可以看到没有之前的与分块相关的参数, 这里进行的运算直接是将整个矩阵进行矩阵运算进行迭代, 算出最后的收敛值。然后就是向rold.dat文件写入每个节点 PageRank 初始值。这里rold文件只有rold.dat一个, 每次迭代计算完的 PageRank 新值将覆盖里面的旧值。

具体运算如下所示,

BasicPagerank/pagerank.cpp

```

193     while (realIterCount < maxIterCount)
194     {
195         cout << "Iteration: " << realIterCount << endl;
196         double leaked = 0.0; // 用于计算 Leaked pagerank
197         double temprank = 0.0;
198         rnew.resize(allNodeCount, 0.0);
199         readMatrix.open("matrix.txt");
200         int src = 0, degree = 0;
201         while (readMatrix >> src >> degree)
202         {
203             int dst = 0, size = degree;
204             rold.seekg(src*sizeof(double), ios::beg);
205             rold.read((char*)&temprank, sizeof(double));
206             while (size)
207             {
208                 readMatrix >> dst;
209                 rnew[dst] += beta*temprank / (double)degree;
210                 size--;
211             }
212         }
213         for (int j = 0; j < rnew.size(); j++)
214         {
215             leaked += rnew[j];
216         }
217         readMatrix.close();
218
219         // 计算完之后将 Leaked pagerank 加回来
220     #ifdef DEBUG

```

```
221     cout << "Calculating leaked pagerank....." << endl;
222 #endif // DEBUG
223     leaked = (1.0 - leaked) / (double)allNodeCount;
224     for (int j = 0; j < rnew.size(); j++)
225     {
226         rnew[j] += leaked;
227     }
228
229     // 计算 convergence
230 #ifdef DEBUG
231     cout << "Calculating convergence....." << endl;
232 #endif // DEBUG
233     rold.seekg(0, ios::beg);
234     dvalue = 0.0;
235     int i = 0;
236     while (true)
237     {
238         double old;
239         rold.read((char*)&old, sizeof(double));
240         if (rold.eof())
241             break;
242         dvalue += fabs(rnew[i] - old);
243         i++;
244     }
245
246     // 更新 rold
247     rold.close();
248     rold.open("rold.dat", ios::in | ios::out | ios::binary | ios::trunc);
249     rold.seekp(0, ios::beg);
250     rold.seekg(0, ios::beg);
251     for (int j = 0; j < rnew.size(); j++)
252     {
253         rold.write((char*)&rnew[j], sizeof(double));
254     }
255
256     realIterCount++;
257     rnew.clear();
258
259     // 判断是否收敛
260     cout << "Convergence: " << dvalue << endl;
```

```
261         if (dvalue <= epsilon)
262             break;
263     }
264     rold.close();
```

依照是否小于等于限制的`epsilon`值作为最外层循环提前退出的条件, 当计算值大于限制的`epsilon`值时, 持续进行计算, 而计算就是每次迭代从`matrix.txt`读入所有数据和从`rold.dat`文件当前源节点的 PageRank, 按照公式计算, 向`rold.dat`文件写入本次迭代的新值, 最后判断是否收敛即可。当然, 此时`rnew`的大小就等于`allNodeCount`了, 存在`vecter`结构中, 一次迭代的过程中不需要反复从`rold`文件中读取。

4 运行过程及云主机截图

本次实验的运行我们使用到了阿里云的服务器, 其系统为 CentOS 7。在 Ubuntu 16.04 系统中使用 `ssh` 登录。

为了更好地分析实验结果, 本次实验通过改变 `teleport` 参数 β 、迭代终止条件 ϵ 值和分块的大小三个参数, 对使用分块的程序进行了七次运行。此外, 针对基础的、只优化了稀疏矩阵而没有分块的程序也运行了一次, 用来和分块版本的程序进行对比。下面是八次运行的结果截图:

4.1 参数值默认

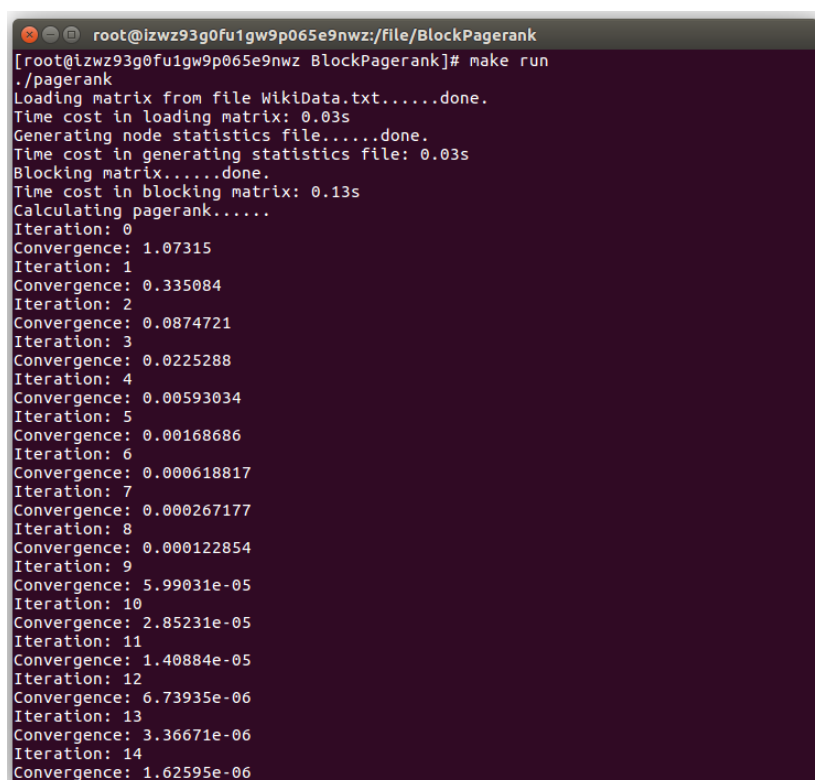
在默认情况下运行程序: $\beta = 0.85$, $\epsilon = 1 \times 10^{-9}$, 每个块最多 100 个节点。后面几次程序运行所选择的参数以此为基准。此时不需要任何参数, 直接输入指令 `make run` 运行, 如下图所示。

通过图 31 和图 32 可以了解到本次 PageRank 算法的执行流程和各部分耗时:

1. 首先读取原始数据集文件 `WikiData.txt`, 生成矩阵, 用时 0.03 秒;
2. 统计节点信息, 生成 `node_statistics.txt` 文件, 用时 0.03 秒;
3. 矩阵分块, 用时 0.13 秒;
4. 迭代对各节点的 PageRank 值进行计算, 一共迭代了 26 次, 用时 1.89 秒, 内存开销 7.05469MB;

5. 将结果写入`result_all.txt`和`result_top100.txt`文件, 它们分别记录了所有节点的 PageRank 值以及其中值最高的 100 个节点, 在这两个文件中, 结果均以[节点ID] [PageRank值]的方式进行记录, 该步用时 0.31 秒。

最后总结来看, 可以了解到: 本次运行的总体时间开销为 2.39 秒, 空间开销 7.16406MB。图 32 中最后的命令 `ls -l blocks | grep "^-" | wc -l` 用来统计分块文件的个数, 即矩阵被分成了多少块, 在本次运行中一共生成了 72 个块。



```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPageRank
[root@izwz93g0fu1gw9p065e9nwz BlockPageRank]# make run
./pagerank
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.03s
Blocking matrix.....done.
Time cost in blocking matrix: 0.13s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
Iteration: 12
Convergence: 6.73935e-06
Iteration: 13
Convergence: 3.36671e-06
Iteration: 14
Convergence: 1.62595e-06
```

Figure 8: 运行过程 1

```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 15
Convergence: 8.11404e-07
Iteration: 16
Convergence: 3.95849e-07
Iteration: 17
Convergence: 1.96826e-07
Iteration: 18
Convergence: 9.68908e-08
Iteration: 19
Convergence: 4.80222e-08
Iteration: 20
Convergence: 2.3826e-08
Iteration: 21
Convergence: 1.17844e-08
Iteration: 22
Convergence: 5.88263e-09
Iteration: 23
Convergence: 2.90481e-09
Iteration: 24
Convergence: 1.46067e-09
Iteration: 25
Convergence: 7.19871e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 1.89s
Real iteration count: 26
Calculate memory cost: 7.05469 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.31s
Total time cost: 2.39s
Total memory cost: 7.16406 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# ls -l blocks | grep "^." | wc -l
72

```

Figure 9: 运行过程 2

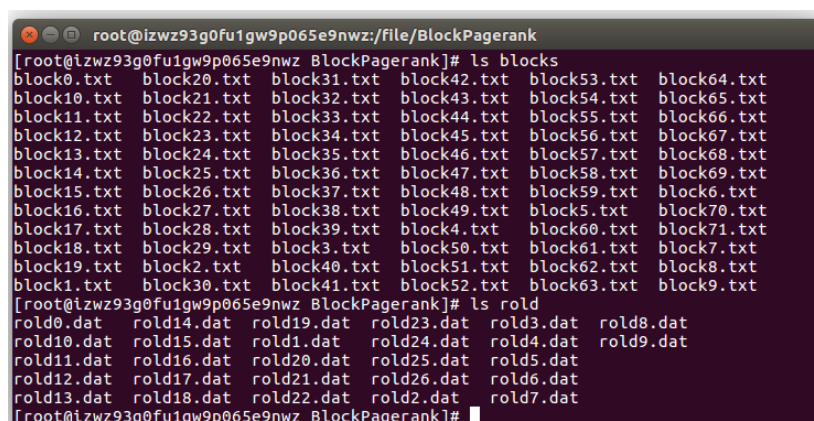
```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
72
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00460717]
[15] [0.00367986]
[6634] [0.00358685]
[2625] [0.00328366]
[2398] [0.00260864]
[2470] [0.00252377]
[2237] [0.00249663]
[4191] [0.00226785]
[7553] [0.00216973]
[5254] [0.0021501]
[2328] [0.00203926]
[1186] [0.00203553]
[1297] [0.00194584]
[4335] [0.00193676]
[7620] [0.00193208]
[5412] [0.00191892]
[7632] [0.00190774]
[4875] [0.00187381]
[6946] [0.00180842]
[3352] [0.00178396]
[6832] [0.00176818]
[2654] [0.00176698]
[762] [0.00174215]
[737] [0.00173963]
[2066] [0.0017157]
[8293] [0.00170531]
[3089] [0.00170201]
[28] [0.00168881]
[2535] [0.0016662]

```

Figure 10: 运行结果

此外，下图为本次执行过程中生成的块文件，共 72 个，以及记录每次迭代各节点 PageRank 值的文件 rold。



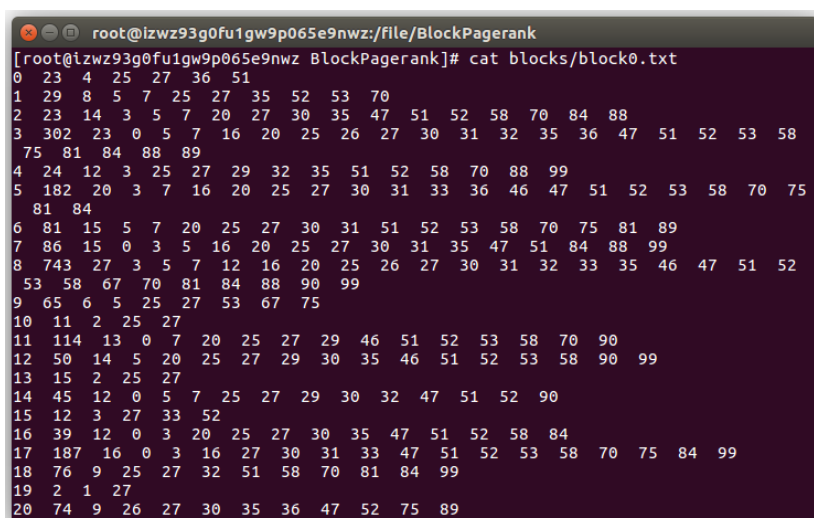
```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# ls blocks
block0.txt  block20.txt  block31.txt  block42.txt  block53.txt  block64.txt
block10.txt  block21.txt  block32.txt  block43.txt  block54.txt  block65.txt
block11.txt  block22.txt  block33.txt  block44.txt  block55.txt  block66.txt
block12.txt  block23.txt  block34.txt  block45.txt  block56.txt  block67.txt
block13.txt  block24.txt  block35.txt  block46.txt  block57.txt  block68.txt
block14.txt  block25.txt  block36.txt  block47.txt  block58.txt  block69.txt
block15.txt  block26.txt  block37.txt  block48.txt  block59.txt  block70.txt
block16.txt  block27.txt  block38.txt  block49.txt  block5.txt  block71.txt
block17.txt  block28.txt  block39.txt  block4.txt  block60.txt  block72.txt
block18.txt  block29.txt  block3.txt  block50.txt  block61.txt  block73.txt
block19.txt  block2.txt  block40.txt  block51.txt  block62.txt  block74.txt
block1.txt  block30.txt  block41.txt  block52.txt  block63.txt  block75.txt
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# ls rold
rold0.dat  rold14.dat  rold19.dat  rold23.dat  rold3.dat  rold8.dat
rold10.dat  rold15.dat  rold1.dat  rold24.dat  rold4.dat  rold9.dat
rold11.dat  rold16.dat  rold20.dat  rold25.dat  rold5.dat
rold12.dat  rold17.dat  rold21.dat  rold26.dat  rold6.dat
rold13.dat  rold18.dat  rold22.dat  rold2.dat  rold7.dat

```

Figure 11: 本次执行过程中生成的块文件和每次迭代的向量文件

块文件**block0.txt**如下图所示，每行表示一个节点对其他节点的指向关系，前三个数值分别为重新编号的节点 ID、节点重要度、以及它指向的本块中的节点个数，从第四个数值开始就是被其指向的各节点的 ID 号。例如第一行的数据表示，ID 为 0 的节点重要度为 23，在本块，即 0 99 号节点中（块大小为 100），有 4 个节点被 0 号节点指向，分别是 25、27、36 和 51。注意这里说的节点 ID 都是在程序中重新编排的，不是源数据的节点 ID。



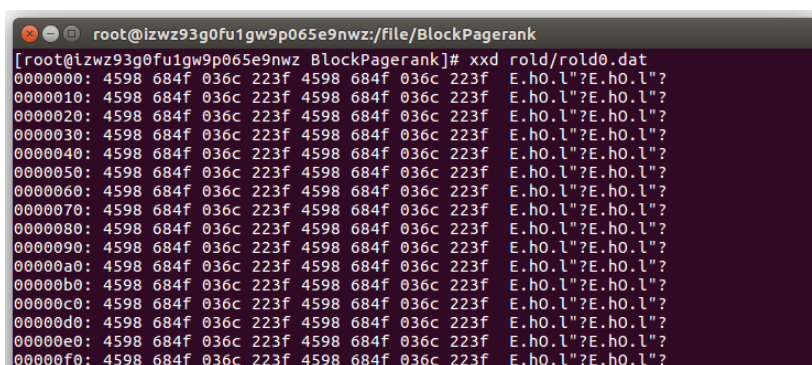
```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat blocks/block0.txt
0 23 4 25 27 36 51
1 29 8 5 7 25 27 35 52 53 70
2 23 14 3 5 7 20 27 30 35 47 51 52 58 70 84 88
3 302 23 0 5 7 16 20 25 26 27 30 31 32 35 36 47 51 52 53 58
75 81 84 88 89
4 24 12 3 25 27 29 32 35 51 52 58 70 88 99
5 182 20 3 7 16 20 25 27 30 31 33 36 46 47 51 52 53 58 70 75
81 84
6 81 15 5 7 20 25 27 30 31 51 52 53 58 70 75 81 89
7 86 15 0 3 5 16 20 25 27 30 31 35 47 51 84 88 99
8 743 27 3 5 7 12 16 20 25 26 27 30 31 32 33 35 46 47 51 52
53 58 67 70 81 84 88 90 99
9 65 6 5 25 27 53 67 75
10 11 2 25 27
11 114 13 0 7 20 25 27 29 46 51 52 53 58 70 90
12 50 14 5 20 25 27 29 30 35 46 51 52 53 58 90 99
13 15 2 25 27
14 45 12 0 5 7 25 27 29 30 32 47 51 52 90
15 12 3 27 33 52
16 39 12 0 3 20 25 27 30 35 47 51 52 58 84
17 187 16 0 3 16 27 30 31 33 47 51 52 53 58 70 75 84 99
18 76 9 25 27 32 51 58 70 81 84 99
19 2 1 27
20 74 9 26 27 30 35 36 47 52 75 89

```

Figure 12: block0.txt

下图展示的是**rold0.dat**文件，其中记录了各节点的初始 PageRank 值，由于其为二进制文件，且程序中使用**double**类型存储该值，因此下图中每 8 个字节对应一个节点的 PageRank 值。可以看出，初始时所有节点的重要度相同。



```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# xxd rold/rold0.dat
00000000: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000010: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000020: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000030: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000040: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000050: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000060: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000070: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000080: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
00000090: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000a0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000b0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000c0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000d0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000e0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?
000000f0: 4598 684f 036c 223f 4598 684f 036c 223f  E.h0.l"?E.h0.l"?

```

Figure 13: rold0.dat

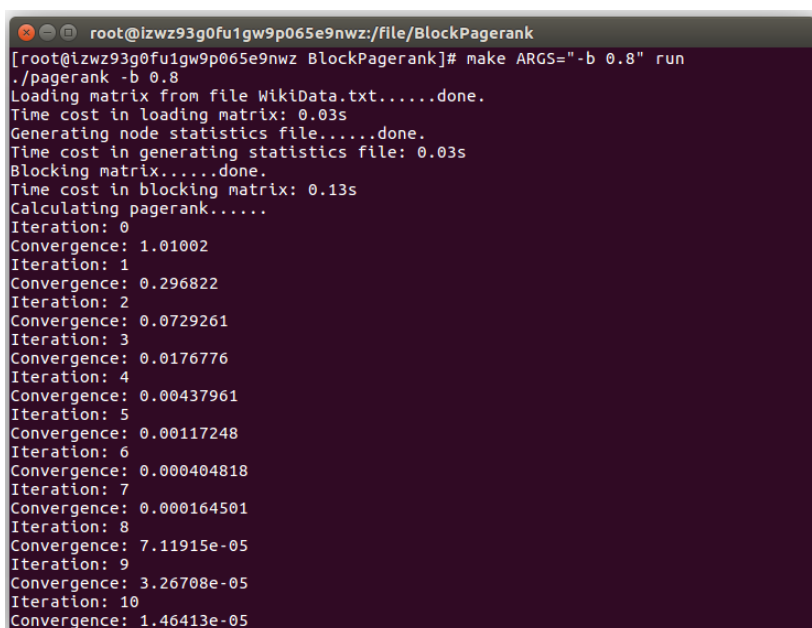
针对此情况生成的结果文件 `result_all.txt` 和 `result_top100.txt` 已经在压缩包中给出。里面的节点 ID 对应源数据的节点 ID，且只统计出现在源文件中的 7115 个节点 ID 的 PageRank 值。

4.2 只改变参数 β

4.2.1 $\beta = 0.8$

改变 β ，使其为 0.8； $\epsilon = 1 \times 10^{-9}$ ，每个块最多 100 个节点两个条件不变。

输入命令 `make ARGS="-b 0.8" run` 运行，这样只会改变参数 β ，而其他两个变量 ϵ 和最大块大小仍然按照默认值—— 1×10^{-9} 和 100 运行。



```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-b 0.8" run
./pagerank -b 0.8
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.03s
Blocking matrix.....done.
Time cost in blocking matrix: 0.13s
Calculating pagerank.....
Iteration: 0
Convergence: 1.01002
Iteration: 1
Convergence: 0.296822
Iteration: 2
Convergence: 0.0729261
Iteration: 3
Convergence: 0.0176776
Iteration: 4
Convergence: 0.00437961
Iteration: 5
Convergence: 0.00117248
Iteration: 6
Convergence: 0.000404818
Iteration: 7
Convergence: 0.000164501
Iteration: 8
Convergence: 7.11915e-05
Iteration: 9
Convergence: 3.26708e-05
Iteration: 10
Convergence: 1.46413e-05

```

Figure 14: 运行过程 1

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 11
Convergence: 6.80636e-06
Iteration: 12
Convergence: 3.06437e-06
Iteration: 13
Convergence: 1.44079e-06
Iteration: 14
Convergence: 6.54899e-07
Iteration: 15
Convergence: 3.07591e-07
Iteration: 16
Convergence: 1.41233e-07
Iteration: 17
Convergence: 6.60938e-08
Iteration: 18
Convergence: 3.06219e-08
Iteration: 19
Convergence: 1.42844e-08
Iteration: 20
Convergence: 6.67027e-09
Iteration: 21
Convergence: 3.10507e-09
Iteration: 22
Convergence: 1.45883e-09
Iteration: 23
Convergence: 6.77991e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 1.83s
Real iteration count: 24
Calculate memory cost: 7.10547 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.32s
Total time cost: 2.34s
Total memory cost: 7.42188 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#
```

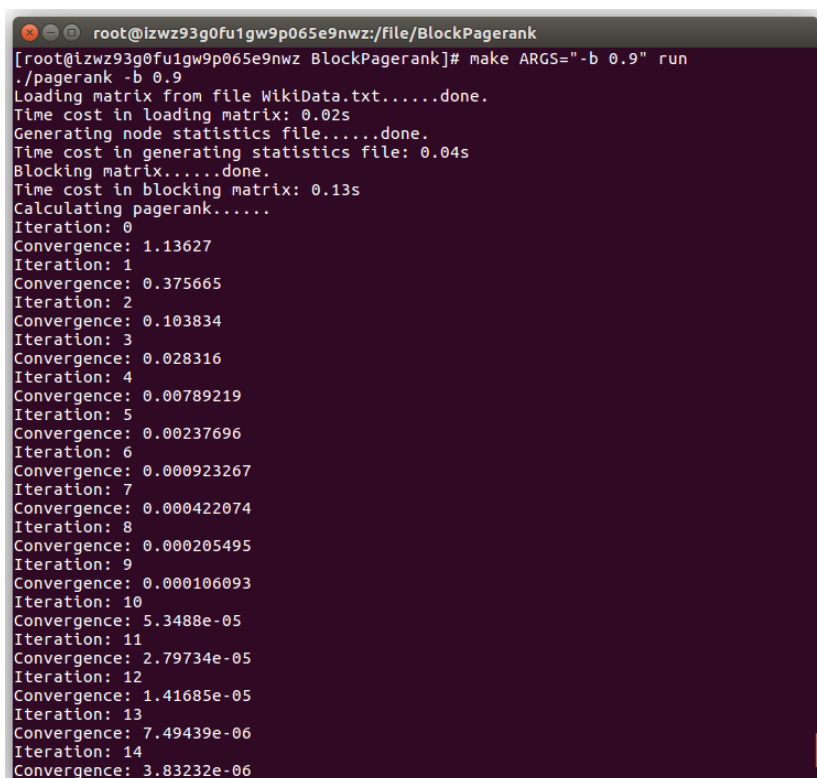
Figure 15: 运行过程 2

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Total memory cost: 7.42188 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00451539]
[15] [0.00354166]
[6634] [0.0032586]
[2625] [0.00311145]
[2470] [0.00253076]
[2237] [0.00247462]
[2398] [0.00244721]
[4191] [0.00216672]
[5254] [0.0020652]
[7553] [0.0020503]
[1186] [0.0020337]
[2328] [0.0019518]
[7620] [0.00184435]
[1297] [0.00183765]
[4335] [0.00181497]
[4875] [0.00179851]
[7632] [0.00177553]
[5412] [0.00177241]
[2654] [0.00173013]
[3352] [0.00169672]
[8293] [0.00168969]
[6832] [0.00165989]
[28] [0.00165487]
[762] [0.00165429]
[665] [0.00164691]
[6946] [0.00163307]
[737] [0.00162994]
[214] [0.00162328]
[6774] [0.00160445]
[2535] [0.00159464]
```

Figure 16: 运行结果

4.2.2 $\beta = 0.9$

改变 β , 使其为 0.9; $\epsilon = 1 \times 10^{-9}$, 每个块最多 100 个节点两个条件不变。输入命令 `make ARGS="-b 0.9" run` 运行。



```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-b 0.9" run
./pagerank -b 0.9
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.02s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.04s
Blocking matrix.....done.
Time cost in blocking matrix: 0.13s
Calculating pagerank.....
Iteration: 0
Convergence: 1.13627
Iteration: 1
Convergence: 0.375665
Iteration: 2
Convergence: 0.103834
Iteration: 3
Convergence: 0.028316
Iteration: 4
Convergence: 0.00789219
Iteration: 5
Convergence: 0.00237696
Iteration: 6
Convergence: 0.000923267
Iteration: 7
Convergence: 0.000422074
Iteration: 8
Convergence: 0.000205495
Iteration: 9
Convergence: 0.000106093
Iteration: 10
Convergence: 5.3488e-05
Iteration: 11
Convergence: 2.79734e-05
Iteration: 12
Convergence: 1.41685e-05
Iteration: 13
Convergence: 7.49439e-06
Iteration: 14
Convergence: 3.83232e-06
```

Figure 17: 运行过程 1

```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 15
Convergence: 2.02495e-06
Iteration: 16
Convergence: 1.046e-06
Iteration: 17
Convergence: 5.50689e-07
Iteration: 18
Convergence: 2.87032e-07
Iteration: 19
Convergence: 1.50631e-07
Iteration: 20
Convergence: 7.9131e-08
Iteration: 21
Convergence: 4.14408e-08
Iteration: 22
Convergence: 2.19035e-08
Iteration: 23
Convergence: 1.14521e-08
Iteration: 24
Convergence: 6.09735e-09
Iteration: 25
Convergence: 3.18176e-09
Iteration: 26
Convergence: 1.70349e-09
Iteration: 27
Convergence: 8.86331e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 2.07s
Real iteration count: 28
Calculate memory cost: 7.10547 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.32s
Total time cost: 2.58s
Total memory cost: 7.42188 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#

```

Figure 18: 运行过程 2

```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Total memory cost: 7.42188 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00468003]
[6634] [0.00395283]
[15] [0.00380942]
[2625] [0.00345569]
[2398] [0.00277401]
[2237] [0.00250494]
[2470] [0.00249748]
[4191] [0.00236783]
[7553] [0.00228606]
[5254] [0.00223128]
[2328] [0.00212469]
[5412] [0.00207251]
[4335] [0.00206221]
[1297] [0.00205337]
[7632] [0.00204385]
[1186] [0.00202413]
[7620] [0.00201808]
[6946] [0.00200466]
[4875] [0.0019478]
[6832] [0.00187813]
[3352] [0.00187025]
[737] [0.00185165]
[2066] [0.00184393]
[762] [0.00182899]
[3089] [0.00181267]
[2654] [0.00179722]
[3334] [0.00173735]
[2535] [0.00173621]

```

Figure 19: 运行结果

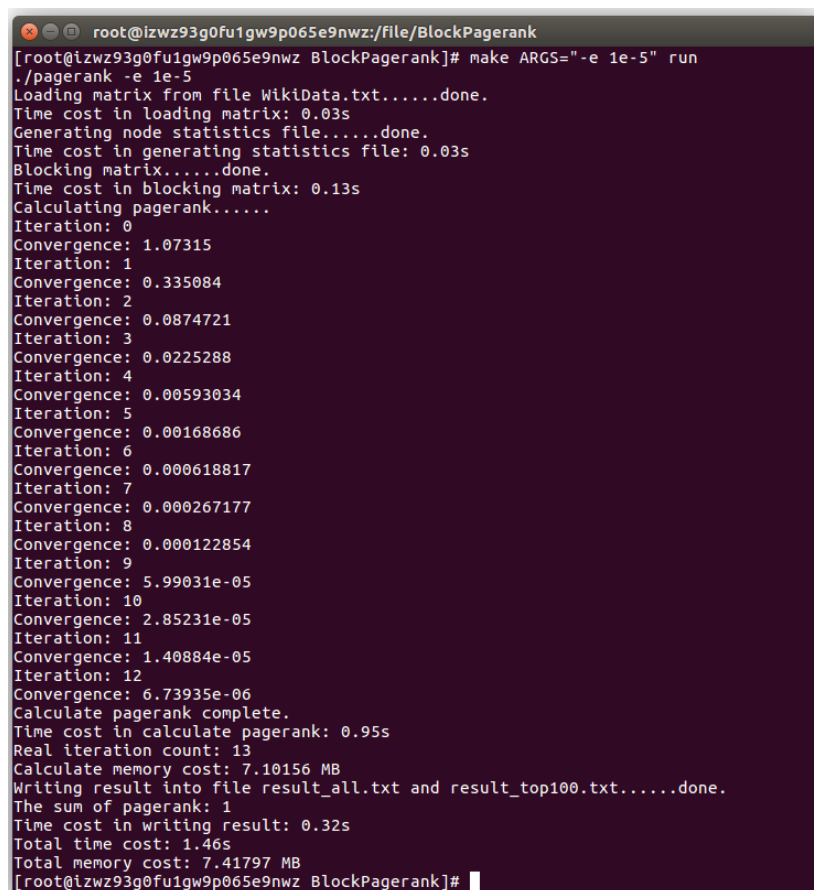
可以看出只改变 β 结果与默认条件下 ($\beta = 0.85$) 的运行结果有偏差。包括每次迭代的 Convergence、迭代次数、结果文件中每个节点的 PageRank 值。

4.3 只改变参数 ϵ

4.3.1 $\epsilon = 1 \times 10^{-5}$

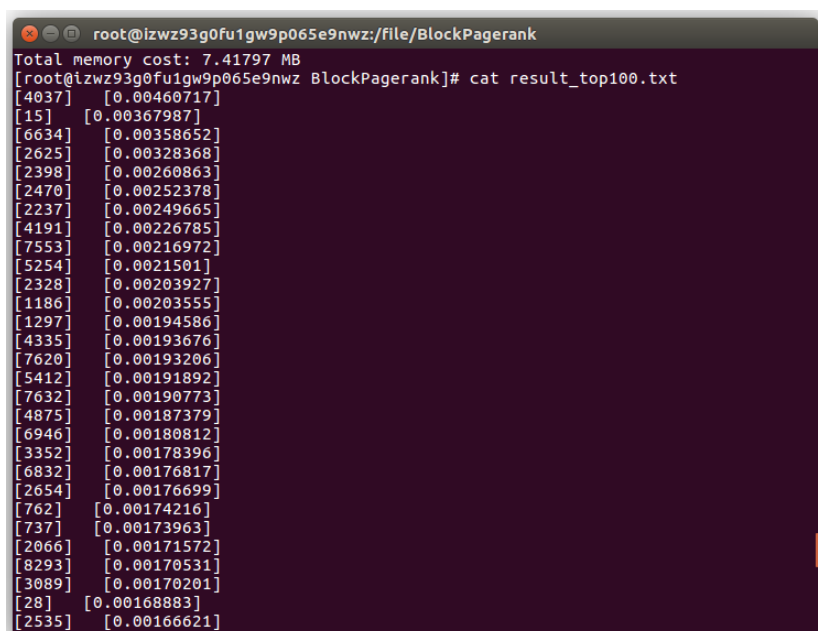
改变 ϵ , 使其为 1×10^{-5} ; $\beta = 0.85$, 每个块最大节点数 100 两个条件不变。

输入命令 `make ARGS="-e 1e-5" run` 运行。



```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-e 1e-5" run
./pagerank -e 1e-5
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.03s
Blocking matrix.....done.
Time cost in blocking matrix: 0.13s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
Iteration: 12
Convergence: 6.73935e-06
Calculate pagerank complete.
Time cost in calculate pagerank: 0.95s
Real iteration count: 13
Calculate memory cost: 7.10156 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.32s
Total time cost: 1.46s
Total memory cost: 7.41797 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#
```

Figure 20: 运行过程



```

root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Total memory cost: 7.41797 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00460717]
[15] [0.00367987]
[6634] [0.00358652]
[2625] [0.00328368]
[2398] [0.00260863]
[2470] [0.00252378]
[2237] [0.00249665]
[4191] [0.00226785]
[7553] [0.00216972]
[5254] [0.0021501]
[2328] [0.00203927]
[1186] [0.00203555]
[1297] [0.00194586]
[4335] [0.00193676]
[7620] [0.00193206]
[5412] [0.00191892]
[7632] [0.00190773]
[4875] [0.00187379]
[6946] [0.00180812]
[3352] [0.00178396]
[6832] [0.00176817]
[2654] [0.00176699]
[762] [0.00174216]
[737] [0.00173963]
[2066] [0.00171572]
[8293] [0.00170531]
[3089] [0.00170201]
[28] [0.00168883]
[2535] [0.00166621]

```

Figure 21: 运行结果

上图对比图 31 和图 32

可以看出, 本次运行过程中 PageRank 计算的迭代次数只有 13 次, 与默认情况下 (1×10^{-9}) 迭代 26 次相比明显减少。

4.3.2 $\epsilon = 1 \times 10^{-7}$

改变 ϵ , 使其为 1×10^{-7} ; $\beta = 0.85$, 每个块最大节点数 100 两个条件不变。

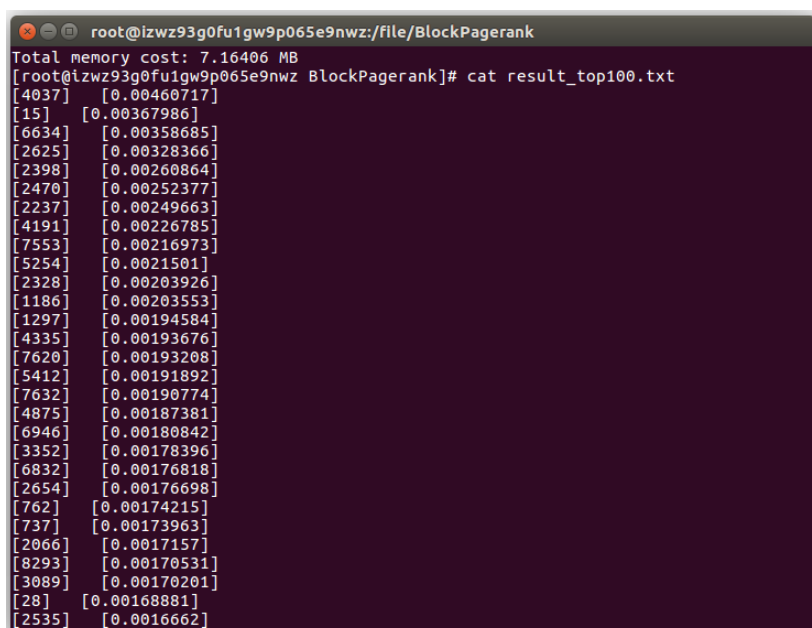
输入命令 `make ARGS=" -e 1e-7" run` 运行。

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-e 1e-7" run
./pagerank -e 1e-7
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.03s
Blocking matrix.....done.
Time cost in blocking matrix: 0.12s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
```

Figure 22: 运行过程 1

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 12
Convergence: 6.73935e-06
Iteration: 13
Convergence: 3.36671e-06
Iteration: 14
Convergence: 1.62595e-06
Iteration: 15
Convergence: 8.11404e-07
Iteration: 16
Convergence: 3.95849e-07
Iteration: 17
Convergence: 1.96826e-07
Iteration: 18
Convergence: 9.68908e-08
Calculate pagerank complete.
Time cost in calculate pagerank: 1.34s
Real iteration count: 19
Calculate memory cost: 7.09375 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.33s
Total time cost: 1.85s
Total memory cost: 7.16406 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#
```

Figure 23: 运行过程 2



```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Total memory cost: 7.16406 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00460717]
[15] [0.00367986]
[6634] [0.00358685]
[2625] [0.00328366]
[2398] [0.00260864]
[2470] [0.00252377]
[2237] [0.00249663]
[4191] [0.00226785]
[7553] [0.00216973]
[5254] [0.0021501]
[2328] [0.00203926]
[1186] [0.00203553]
[1297] [0.00194584]
[4335] [0.00193676]
[7620] [0.00193208]
[5412] [0.00191892]
[7632] [0.00190774]
[4875] [0.00187381]
[6946] [0.00180842]
[3352] [0.00178396]
[6832] [0.00176818]
[2654] [0.00176698]
[762] [0.00174215]
[737] [0.00173963]
[2066] [0.0017157]
[8293] [0.00170531]
[3089] [0.00170201]
[28] [0.00168881]
[2535] [0.0016662]
```

Figure 24: 运行结果

可以看出只改变 ϵ 结果与默认条件 ($\epsilon = 1 \times 10^{-9}$) 相比, 每次迭代的 Convergence 不会变化, 迭代次数会发生变化。相应地, 结果文件中每个节点的 PageRank 值也会发生变化。

4.4 只改变分块大小

4.4.1 改变分块大小为 50

令每个块最多有 50 个节点; $\beta = 0.85$, $\epsilon = 1 \times 10^{-9}$ 两个条件不变。

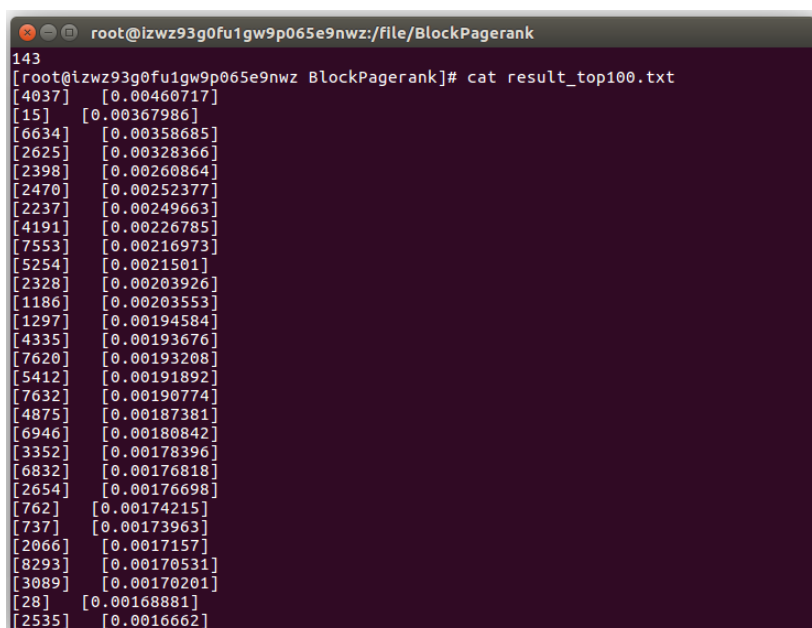
输入命令 `make ARGS="-n 50" run` 运行。


```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-n 50" run
./pagerank -n 50
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.02s
Blocking matrix.....done.
Time cost in blocking matrix: 0.15s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
Iteration: 12
Convergence: 6.73935e-06
```

Figure 25: 运行过程 1

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 13
Convergence: 3.36671e-06
Iteration: 14
Convergence: 1.62595e-06
Iteration: 15
Convergence: 8.11404e-07
Iteration: 16
Convergence: 3.95849e-07
Iteration: 17
Convergence: 1.96826e-07
Iteration: 18
Convergence: 9.68908e-08
Iteration: 19
Convergence: 4.80222e-08
Iteration: 20
Convergence: 2.3826e-08
Iteration: 21
Convergence: 1.17844e-08
Iteration: 22
Convergence: 5.88263e-09
Iteration: 23
Convergence: 2.90481e-09
Iteration: 24
Convergence: 1.46067e-09
Iteration: 25
Convergence: 7.19871e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 2.27s
Real iteration count: 26
Calculate memory cost: 7.10547 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.32s
Total time cost: 2.79s
Total memory cost: 7.67969 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# ls -l blocks | grep "^-" | wc -l
143
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#
```

Figure 26: 运行过程 2



```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
143
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00460717]
[15] [0.00367986]
[6634] [0.00358685]
[2625] [0.00328366]
[2398] [0.00260864]
[2470] [0.00252377]
[2237] [0.00249663]
[4191] [0.00226785]
[7553] [0.00216973]
[5254] [0.0021501]
[2328] [0.00203926]
[1186] [0.00203553]
[1297] [0.00194584]
[4335] [0.00193676]
[7620] [0.00193208]
[5412] [0.00191892]
[7632] [0.00190774]
[4875] [0.00187381]
[6946] [0.00180842]
[3352] [0.00178396]
[6832] [0.00176818]
[2654] [0.00176698]
[762] [0.00174215]
[737] [0.00173963]
[2066] [0.0017157]
[8293] [0.00170531]
[3089] [0.00170201]
[28] [0.00168881]
[2535] [0.0016662]
```

Figure 27: 运行结果

4.4.2 改变分块大小为 500

令每个块最多有 500 个节点; $\beta = 0.85$, $\epsilon = 1 \times 10^{-9}$ 两个条件不变。
输入命令 `make ARGS="-n 500" run` 运行。

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# make ARGS="-n 500" run
./pagerank -n 500
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.03s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.03s
Blocking matrix.....done.
Time cost in blocking matrix: 0.08s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
Iteration: 12
Convergence: 6.73935e-06
```

Figure 28: 运行过程 1

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
Iteration: 13
Convergence: 3.36671e-06
Iteration: 14
Convergence: 1.62595e-06
Iteration: 15
Convergence: 8.11404e-07
Iteration: 16
Convergence: 3.95849e-07
Iteration: 17
Convergence: 1.96826e-07
Iteration: 18
Convergence: 9.68908e-08
Iteration: 19
Convergence: 4.80222e-08
Iteration: 20
Convergence: 2.3826e-08
Iteration: 21
Convergence: 1.17844e-08
Iteration: 22
Convergence: 5.88263e-09
Iteration: 23
Convergence: 2.90481e-09
Iteration: 24
Convergence: 1.46067e-09
Iteration: 25
Convergence: 7.19871e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 1.21s
Real iteration count: 26
Calculate memory cost: 7.07812 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.31s
Total time cost: 1.66s
Total memory cost: 7.125 MB
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# ls -l blocks | grep "^-" | wc -l
15
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]#
```

Figure 29: 运行过程 2

```
root@izwz93g0fu1gw9p065e9nwz:/file/BlockPagerank
15
[root@izwz93g0fu1gw9p065e9nwz BlockPagerank]# cat result_top100.txt
[4037] [0.00460717]
[15] [0.00367986]
[6634] [0.00358685]
[2625] [0.00328366]
[2398] [0.00260864]
[2470] [0.00252377]
[2237] [0.00249663]
[4191] [0.00226785]
[7553] [0.00216973]
[5254] [0.0021501]
[2328] [0.00203926]
[1186] [0.00203553]
[1297] [0.00194584]
[4335] [0.00193676]
[7620] [0.00193208]
[5412] [0.00191892]
[7632] [0.00190774]
[4875] [0.00187381]
[6946] [0.00180842]
[3352] [0.00178396]
[6832] [0.00176818]
[2654] [0.00176698]
[762] [0.00174215]
[737] [0.00173963]
[2066] [0.0017157]
[8293] [0.00170531]
[3089] [0.00170201]
[28] [0.00168881]
[2535] [0.0016662]
```

Figure 30: 运行结果

4.5 运行只优化稀疏矩阵的程序

最后，对只优化了稀疏矩阵，而没有分块的程序运行一次。

在BasicPagerank文件夹下输入命令`make run`，以默认的参数设置运行，其中 $\beta = 0.85$ ， $\epsilon = 1 \times 10^{-9}$ 。

```
root@izwz93g0fu1gw9p065e9nwz:/file/BasicPagerank
[root@izwz93g0fu1gw9p065e9nwz BasicPagerank]# make run
./pagerank
Loading matrix from file WikiData.txt.....done.
Time cost in loading matrix: 0.08s
Generating node statistics file.....done.
Time cost in generating statistics file: 0.04s
Calculating pagerank.....
Iteration: 0
Convergence: 1.07315
Iteration: 1
Convergence: 0.335084
Iteration: 2
Convergence: 0.0874721
Iteration: 3
Convergence: 0.0225288
Iteration: 4
Convergence: 0.00593034
Iteration: 5
Convergence: 0.00168686
Iteration: 6
Convergence: 0.000618817
Iteration: 7
Convergence: 0.000267177
Iteration: 8
Convergence: 0.000122854
Iteration: 9
Convergence: 5.99031e-05
Iteration: 10
Convergence: 2.85231e-05
Iteration: 11
Convergence: 1.40884e-05
Iteration: 12
Convergence: 6.73935e-06
Iteration: 13
Convergence: 3.36671e-06
Iteration: 14
Convergence: 1.62595e-06
```

Figure 31: 运行过程 1

```
root@izwz93g0fu1gw9p065e9nwz:/file/BasicPagerank
Iteration: 15
Convergence: 8.11404e-07
Iteration: 16
Convergence: 3.95849e-07
Iteration: 17
Convergence: 1.96826e-07
Iteration: 18
Convergence: 9.68908e-08
Iteration: 19
Convergence: 4.80222e-08
Iteration: 20
Convergence: 2.3826e-08
Iteration: 21
Convergence: 1.17844e-08
Iteration: 22
Convergence: 5.88263e-09
Iteration: 23
Convergence: 2.90481e-09
Iteration: 24
Convergence: 1.46067e-09
Iteration: 25
Convergence: 7.19871e-10
Calculate pagerank complete.
Time cost in calculate pagerank: 0.46s
Real iteration count: 26
Calculate memory cost: 6.96875 MB
Writing result into file result_all.txt and result_top100.txt.....done.
The sum of pagerank: 1
Time cost in writing result: 0.32s
Total time cost: 0.9s
Total memory cost: 6.99219 MB
[root@izwz93g0fu1gw9p065e9nwz BasicPagerank]#
```

Figure 32: 运行过程 2

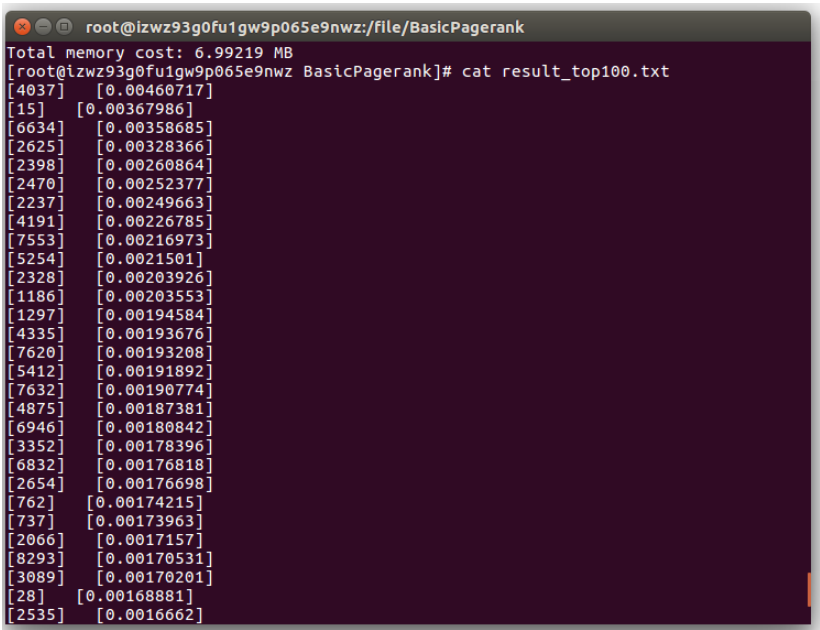


Figure 33: 运行结果

5 实验结果及分析

按照第 4 节的实验截图，我们会分析 teleport 参数 β 、迭代终止条件 ϵ 值、分块的大小三个参数分别改变时，对 PageRank 的运行产生的影响。具体方法为：以默认条件（ $\beta = 0.85$ ， $\epsilon = 1 \times 10^{-9}$ ，每个块最多 100 个节点）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。

5.1 teleport 参数 β 对程序运行结果的影响

保持 $\epsilon = 1 \times 10^{-9}$ ，每个块最多 100 个节点，使 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	块大小	迭代次数	时间消耗	空间消耗	块总数
1	0.8	1.00E-09	100	24	2.34 s	7.42188 MB	72
2	0.85	1.00E-09	100	26	2.39 s	7.16406MB	72
3	0.9	1.00E-09	100	28	2.58 s	7.42188 MB	72

Table 2: teleport 参数 β 对程序运行结果的影响

可以看出, β 值的变化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, 且块总数也因为块大小没有改变而保持不变。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, 如下图所示。

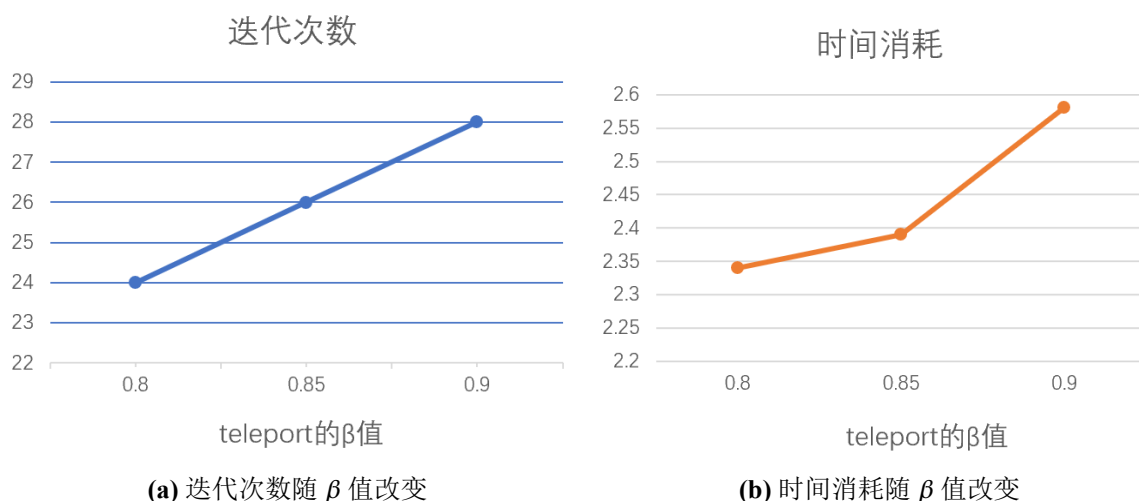


Figure 34: 迭代次数和时间消耗随 β 值改变

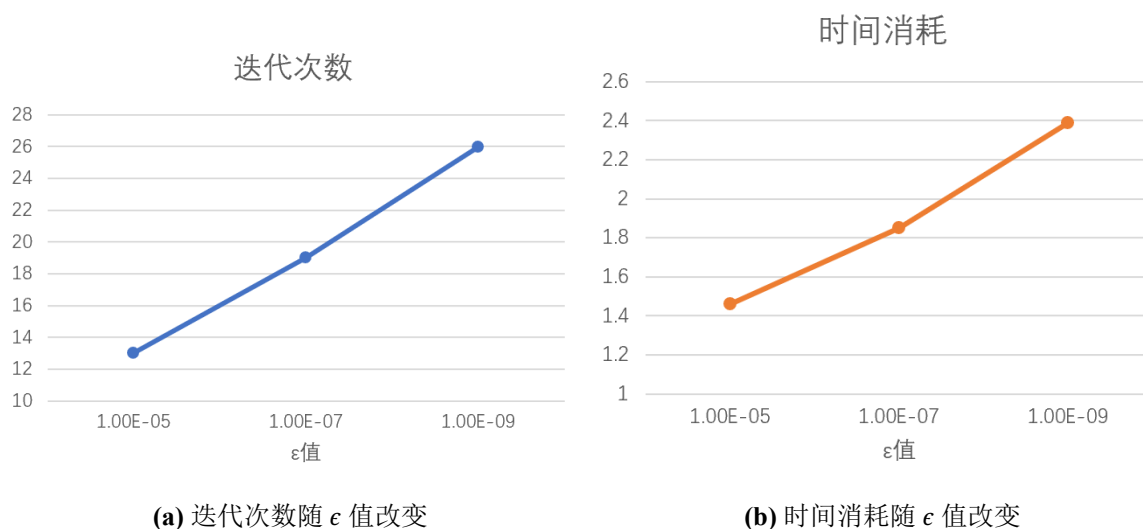
5.2 迭代终止条件 ϵ 对程序运行结果的影响

保持 $\beta = 0.85$, 每个块最多 100 个节点不变, 使 β 分别为 1×10^{-5} 、 1×10^{-7} 、 1×10^{-9} 时, 结果分析如下:

运行参数				运行结果			
序号	β	ϵ	块大小	迭代次数	时间消耗	空间消耗	块总数
1	0.85	1.00E-05	100	13	1.46 s	7.41797 MB	72
2	0.85	1.00E-07	100	19	1.85 s	7.16406MB	72
3	0.85	1.00E-09	100	26	2.39 s	7.16406MB	72

Table 3: 迭代终止条件 ϵ 改变对程序运行结果的影响

可以看出, 随着 β 不断减小, 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, 运行时间也会相应地增加, 如下图所示。而 ϵ 改变对于空间消耗并不会产生很大影响, 且由于块总数只受块大小的影响, 因此也不会改变。

Figure 35: 迭代次数和时间消耗随 ϵ 值改变

5.3 分块的大小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、块大小为 50、100、500 时的运行结果, 如下表所示:

序号	运行参数			运行结果			
	β	ϵ	块大小	迭代次数	时间消耗	空间消耗	块总数
1	0.85	1.00E-09	/	26	0.9 s	6.99219 MB	/
2	0.85	1.00E-09	50	26	2.79 s	7.67969 MB	143
3	0.85	1.00E-09	100	26	2.39 s	7.16406 MB	72
4	0.85	1.00E-09	500	26	1.66 s	7.125 MB	15

Table 4: 块大小改变对程序运行结果的影响

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。

改变分块大小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 `rold` 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`roid`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`roid`文件次数变少), 并逐渐逼近不分块的情况下的时间开销, 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。

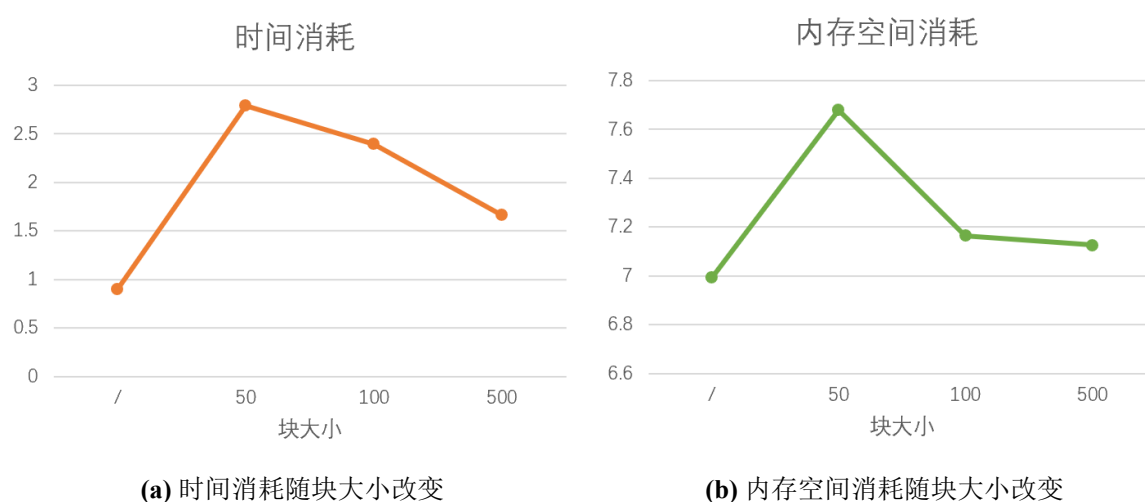


Figure 36: 时间消耗和内存空间消耗随块大小改变

5.4 总结

1. 对于以上几次运行的结果, 可以看出, 微调参数 β 、 ϵ 不会对结果文件 `result_all.txt` 和 `result_top100.txt` 中节点 ID 的排序产生太大影响, 而影响的主要是迭代次数和计算时间, 还有每个节点的 PageRank 值。改变 β 会导致每次迭代的 Convergence 发生改变; 改变 ϵ 不会导致每次迭代的 Convergence 发生变化, 但是迭代次数会改变。 ϵ 值越小, 收敛条件越严格, 迭代次数就越多, 结果精度就越高。
2. 调整分块大小不会改变迭代次数和生成的结果, 而计算时间和内存用量会有改变, 这也印证了程序当中分块算法的正确性。
3. 由第 2 节可知, 原始数据集是有一些 Dead End 节点的, 而使用了 The Complete

Algorithm, 将每次因为 Dead End 节点所丢失的 PageRank 值加回给每个节点。所以无论有没有 Dead End 节点, 无论如何改变参数, 实验结果中收敛之后所有节点 PageRank 值相加的和总为 1, 这也印证了程序当中实现的 PageRank 基础算法的正确性。

4. 除了时间消耗以外, 我们试图分析内存消耗, 但是却没有得出特别好的结果, 总结原因如下:

- 原始数据集不够大, 体现不出分块算法的优势
- 编译器的优化导致运行时间和内存被优化, 影响对内存消耗的判断
- 进行分块操作时和计算 PageRank 值的时候会额外定义一些临时变量, 再加上每次运行的时候, 进程所处的当前环境也有略微不同, 也会影响对内存消耗的判断
- 测试的时候也许应该选择更合适的参数
- 由于对计算 PageRank 值的内存测量只是简单地测量进程当前占用的物理内存大小, 也许需要更强大的内存分析工具