



南開大學  
Nankai University

计算机学院  
大数据计算及应用

## PageRank 算法实现

小组成员：戴廷钧 高森森 牟迪  
专业：计算机科学与技术

2023 年 4 月 30 日

# 目录

<b>1 实验要求</b>	<b>2</b>
<b>2 数据集说明</b>	<b>2</b>
<b>3 PageRank 算法 (存储至内存)</b>	<b>2</b>
3.1 关于稀疏矩阵的处理	2
3.2 ID 与索引的映射	3
3.3 Dead-ends 与 Spider-trap	3
3.4 迭代计算	4
3.5 排序结果处理	5
3.6 运行结果	6
<b>4 PageRank 算法 (存储至磁盘)</b>	<b>7</b>
4.1 邻接矩阵的存储	7
4.2 迭代计算	8
4.3 运行结果	8
<b>5 Block-based 更新算法</b>	<b>9</b>
5.1 每次仅计算一部分的 $r_{new}$	9
5.2 $\frac{1-S}{N}$ 值处理	10
5.3 运行结果	10
<b>6 Block-Stripe 更新算法</b>	<b>11</b>
6.1 邻接矩阵分块	12
6.2 $r^{old}$ 向量的保存	13
6.3 运行结果	14
<b>7 算法性能与结果分析</b>	<b>15</b>
7.1 各算法时间与内存性能比较	15
7.2 游走参数对算法的影响	16
7.3 分块大小对算法性能的影响	16

## 1 实验要求

基于给定的数据集，实现 pagerank 算法计算各节点 pagerank 分数，最终得到分数前 100 的节点，同时在 pagerank 算法的基础上，实现 Block-Stride Update algorithm。调整算法中传送参数的大小。比较在不同的传送参数下所得的排序结果。

在实现 pagerank 算法的过程中。需要考虑其中的 deadends 节点与 spider trap 节点。需要进行稀疏矩阵的优化，并实现分块计算。最终排序结果取程序迭代至收敛状态后的结果。

## 2 数据集说明

在本次实验中所采用的数据集为作业要求中提供的 Data.txt 文件。其每一行代表图网络中的一条边。格式为：

***FromID toID***

对数据集进行初步的扫描与整理，得如下结论。

1. 图网络中边的总数：数据集中共包含 83852 条边。
2. 节点序号的范围：数据集中共包含 6263 个节点，其中最小节点序号为 3，最大节点序号为 8297
3. dead-ends 节点：dead-ends 节点即网络中出度为 0，但入度不为 0 的节点。通过对数据集的扫描，得到数据集中共存在 767 个 dead-ends 节点。
4. 重复边：数据集中共包含 83852 条关系描述，其中存在 2100 条重复的边。在迭代计算节点的 pagerank 分数时，需要考虑重复的边对于提升链接权重的影响。

在本次实验中，我们分别实现了基础 pagerank 算法 (数据存储在内存中)，基础 pagerank 算法 (数据存储在磁盘中)，Block-based 更新算法与 Block-Stripe 更新算法。并进行了时间与内存性能的比较，分析了不同游走参数下算法收敛情况的分析。

## 3 PageRank 算法 (存储至内存)

### 3.1 关于稀疏矩阵的处理

数据集中节点的总数过大，因此如果使用一般的形式对邻接矩阵进行编码，矩阵中存在较多的 0 项。对于内存空间以及迭代计算来说存在较大开销。在本次的基础 pagerank 算法中，采用稀疏矩阵的形式对邻接矩阵进行编码，只保留矩阵中的非 0 项。在代码实现中，维护一个结构体 NodeData，其中保存了当前节点的出度，并通过 vector 容器来保存当前节点所到达的所有节点。以 map 容器的形式维护每个节点与其对应的 NodeData 的映射。形成最终的稀疏矩阵。关键代码如下：

NodeData 结构体

```
1 struct NodeData {  
2     int degree=0;  
3     vector<int> tos;  
4 };
```

## NodeData 结构体

```

1  map<int, NodeData> m;
2  set<int> nodes;
3  while (getline(infile, line))
4  {
5      stringstream ss(line);
6      int from_, to;
7      ss >> from_ >> to;
8      m[from_].degree++;
9      m[from_].tos.push_back(to);
10     nodes.insert(from_);
11     nodes.insert(to);
12 }
13 for (auto item : m)
14 {
15     NodeData data = item.second;
16     sort(data.tos.begin(), data.tos.end());
17 }
18 vector<int> sorted_nodes(nodes.begin(), nodes.end());

```

### 3.2 ID 与索引的映射

根据数据集分析的情况,在节点序号的范围 0——8297 内存在若干孤立节点,出度与入度为 0,不与其他节点产生联系。在迭代计算中,孤立节点也不会对其他节点的 pagerank 值产生贡献。因此选择以 set 集合保存最终的非孤立节点。即上述稀疏矩阵处理中得到的 nodes 集合。为了便于后续的遍历,将其转换为了 vector 容器 sorted\_nodes,此时在 vector 容器中,每个节点的索引并不对应这节点真实的 ID,因此通过一个 map 映射 id2Serial 来保存节点 ID 与其真实索引之间的关系,关键代码如下:

## id2Serial

```

1  map<int, int> id2Serial;
2  int idx = 0;
3  for (auto id : sorted_nodes)
4  {
5      id2Serial[id] = idx++;
6  }

```

### 3.3 Dead-ends 与 Spider-trap

在 pagerank 算法的学习过程中得,Dead-ends 与 Spider-trap 是影响 pagerank 算法正确性的两个关键问题。spider-trap 节点会使得节点得 pagerank 值不断得在两个或多个节点之间传递。最终使得由两个节点或多个节点组成得一个子网络中得节点的重要程度不断上升,并会使得算法最终无法收敛。而 dead-ends 节点因为其出度为 0. 因此,其 pagerank 值不会贡献给其他节点。使得网络中在传递的 pagerank 值得和不断减小。最终所有节点得 pagerank 值都趋近于 0. 这两种节点都需要进行对应的处理。

对于 spider-trap 节点,其 pagerank 值的传递一直在若干节点内部,因此,在计算中引入游走参

数  $\beta$ , 其可视为在节点得游走过程中, 有  $\beta$  的概率按照真实存在的链接, 随机游走至下一个节点。而有  $1 - \beta$  的概率随机跳转到网络中的任意节点中。从而使得陷入 spider-trap 后, 仍有一定的概率可以返回到原来的网络中。体现在每一个节点的 pagerank 值迭代中为:

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

其中  $(1 - \beta) \frac{1}{N}$  代表了网络中的所有节点都有  $1 - \beta$  的概率对当前节点进行贡献。在网络中没有 dead-ends 节点的情况下, 网络中的 pagerank 值之和为 1, 并均等的贡献给全部的  $N$  个节点。

对于 dead-ends 节点, 同样采用随机游走的思想, 在每次迭代时将 deadends 节点所持有的 pagerank 值均等的贡献给网络中的其他节点, 从而保证了网络中总的 pagerank 值不会损失。在上述公式的基础上进行修正。每一个节点的 pagerank 值迭代中为:

$$r_j = \sum_{i \rightarrow j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N} + \frac{1 - S}{N} \quad (2)$$

$$S = \sum_{degree_i \neq 0} r_i \quad (3)$$

在不存在 dead-ends 节点的情况下,  $S$  值将为 1。

### 3.4 迭代计算

依据公式2, 对所有节点的 pagerank 值进行迭代计算, 直至收敛。关键代码如下:

#### 迭代计算

```

1  for (int iter = 0; iter < max_iter; ++iter)
2  {
3      double init = (1 - beta) / nodes_num;
4      vector<double> r_new(nodes_num, init);
5      int r_index = 0;
6      map<int, NodeData>::iterator iter_Matrix = m.begin();
7      while (iter_Matrix != m.end())
8      {
9          int from_ = (*iter_Matrix).first;
10         int degree = (*iter_Matrix).second.degree;
11         int from_idx = id2Serial[from_];
12         double r_;
13         while (r_index != from_idx)
14         {
15             r_index++;
16         }
17         r_ = r_old[r_index] / degree;
18         r_index++;
19         for (int i = 0; i < degree; ++i)
20         {
21             int to = (*iter_Matrix).second.tos[i];
22             int idx = id2Serial[to];
23             r_new[idx] += beta * r_;

```

```

24     }
25     iter_Matrix++;
26 }
27
28 double r_new_sum = 0;
29 for (const auto& r_ : r_new) {
30     r_new_sum += r_;
31 }
32 double delta = (1 - r_new_sum) / nodes_num;
33 for (auto& r_ : r_new) {
34     r_ += delta;
35 }
36 double err = 0;
37 for (int i = 0; i < r_old.size(); i++)
38 {
39     err += abs(r_old[i] - r_new[i]);
40 }
41 for (int i = 0; i < r_old.size(); i++)
42 {
43     r_old[i] = r_new[i];
44 }
45 if (err < epsilon) {
46     cout << "finish at iter " << iter << endl;
47     break;
48 }
49 }

```

在迭代计算中,因为稀疏邻接矩阵中的每一行只保存了出度不为 0 的节点,体现在代码中即 map 映射中只记录了出度不为 0 的节点与其指向的节点信息。而 sorted\_nodes 中则保存了网络中所有节点。在进行迭代时,需要保证二者指向的是同一个节点。即代码中 11 行至 16 行所完成的工作。

### 3.5 排序结果处理

目前已有的 r 向量中的节点顺序为节点 ID 的升序,需要将其转换为节点 pagerank 值的降序并保留原有的节点 ID 信息。对原有的 vector 容器进行合并与重新排序,并最终输出排序前 100 的节点结果。关键代码如下:

#### 结果处理

```

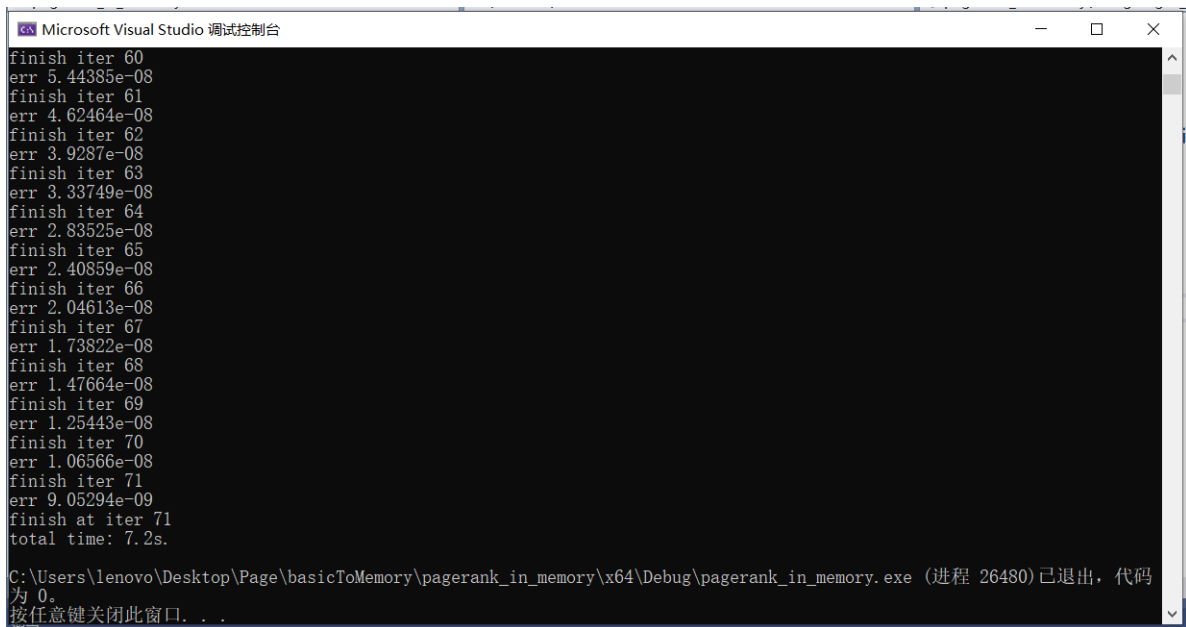
1  vector<pair<int, double>>vec;
2  for (int i = 0; i < r_old.size(); i++)
3  {
4      vec.push_back({ sorted_nodes[i], r_old[i] });
5  }
6  sort(vec.begin(), vec.end(), compare);
7  ofstream result(outfile);
8  int num = 0;
9  for (vector<std::pair<int, double>>::iterator it = vec.begin(); it != vec.end();
    ++it) {

```

```
10     result << it->first << "\t" << setprecision(10) << it->second << "\n";
11     num++;
12
13     if (num == 100) {
14         break;
15     }
16 }
```

### 3.6 运行结果

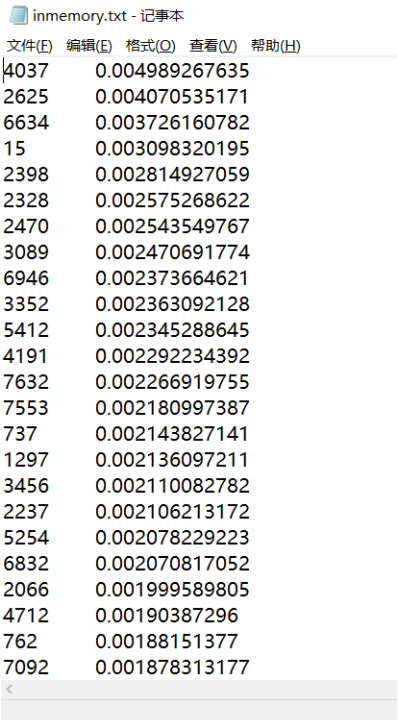
在游走参数  $\beta$  取 0.85, 收敛条件 epsilon 取  $1e-8$  的情况下, 程序运行结果截图如下:



```
Microsoft Visual Studio 调试控制台
finish iter 60
err 5.44385e-08
finish iter 61
err 4.62464e-08
finish iter 62
err 3.9287e-08
finish iter 63
err 3.33749e-08
finish iter 64
err 2.83525e-08
finish iter 65
err 2.40859e-08
finish iter 66
err 2.04613e-08
finish iter 67
err 1.73822e-08
finish iter 68
err 1.47664e-08
finish iter 69
err 1.25443e-08
finish iter 70
err 1.06566e-08
finish iter 71
err 9.05294e-09
finish at iter 71
total time: 7.2s.
C:\Users\lenovo\Desktop\Page\basicToMemory\pagerank_in_memory\x64\Debug\pagerank_in_memory.exe (进程 26480) 已退出, 代码为 0。
按任意键关闭此窗口。...
```

图 3.1: 运行时结果

算法迭代至 71 次时完成收敛, 总用时 7.2 秒, 算法所得排序结果如下:



4037	0.004989267635
2625	0.004070535171
6634	0.003726160782
15	0.003098320195
2398	0.002814927059
2328	0.002575268622
2470	0.002543549767
3089	0.002470691774
6946	0.002373664621
3352	0.002363092128
5412	0.002345288645
4191	0.002292234392
7632	0.002266919755
7553	0.002180997387
737	0.002143827141
1297	0.002136097211
3456	0.002110082782
2237	0.002106213172
5254	0.002078229223
6832	0.002070817052
2066	0.001999589805
4712	0.00190387296
762	0.00188151377
7092	0.001878313177

图 3.2: 排序结果

## 4 PageRank 算法 (存储至磁盘)

在上述介绍中, 实现了将数据存储至内存中 pagerank 算法, pagerank 算法对内存开销较大, 当内存中仅有一定的空间能够支持保存每一次新的  $r^{new}$  向量时, 此时可以将邻接矩阵 Matrix 和  $r^{old}$  保存至磁盘中, 在每次迭代时逐行进行读取。

### 4.1 邻接矩阵的存储

与上述存储至内存的 PageRank 算法相同, 采用 map 结合 NodeData 结构体的形式构建稀疏矩阵。此时将构建好的邻接矩阵逐行输出至磁盘文件中。关键代码如下:

邻接矩阵的存储

```

1  while (getline(infile, line)) {
2      stringstream ss(line);
3      int from_, to;
4      ss >> from_ >> to;
5
6      m[from_].degree++;
7      m[from_].tos.push_back(to);
8  }
9  ofstream outfile(sparse_matrix_file);
10 for (const auto& item : m) {
11     int from_ = item.first;
12     NodeData data = item.second;
13     sort(data.tos.begin(), data.tos.end());

```



```

14     outfile << from_ << " " << data.degree << " ";
15     for (size_t i = 0; i < data.tos.size(); ++i) {
16         outfile << data.tos[i];
17         if (i < data.tos.size() - 1) {
18             outfile << " ";
19         }
20     }
21     outfile << "\n";
22     nodes.insert(from_);
23     for (const auto& to : data.tos) {
24         nodes.insert(to);
25     }
26 }

```

## 4.2 迭代计算

在每次迭代过程中, 需要从磁盘文件中重新读取 Matrix 矩阵和  $r^{old}$  向量, 同样需要注意 Matrix 矩阵和  $r^{old}$  向量的对齐问题。在每次迭代结束之后, 需要直接将  $r^{new}$  保存在磁盘文件中, 替代原有的  $r^{old}$  向量。具体的迭代计算过程与存储值内存的 pagerank 算法相同。

## 4.3 运行结果

在游走参数  $\beta$  取 0.85, 收敛条件 epsilon 取  $1e-8$  的情况下, 程序运行结果截图如下:

```

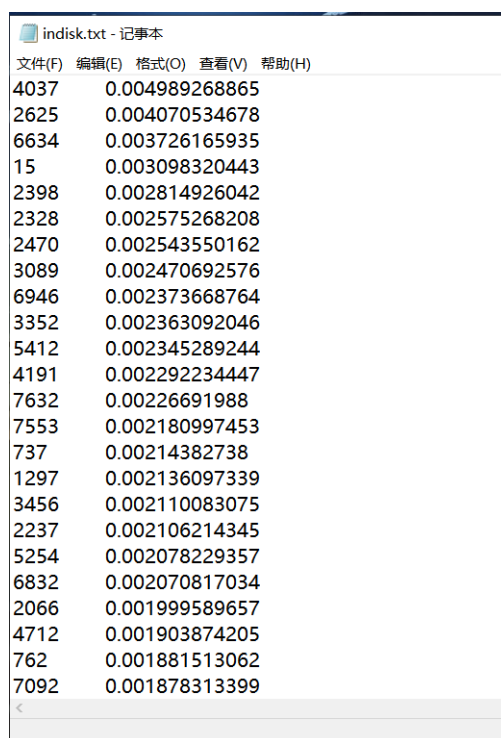
Microsoft Visual Studio 调试控制台
finish iter 58
err 7.2991e-08
finish iter 59
err 5.60227e-08
finish iter 60
err 5.43576e-08
finish iter 61
err 5.43672e-08
finish iter 62
err 5.26406e-08
finish iter 63
err 2.03769e-08
finish iter 64
err 1.86791e-08
finish iter 65
err 1.86813e-08
finish iter 66
err 1.86764e-08
finish iter 67
err 1.69897e-08
finish iter 68
err 1.69867e-08
finish iter 69
err 7.02187e-09
finish at iter 69
total time: 32.22s.

C:\Users\lenovo\Desktop\Page\basicToMemory\pagerank_in_memory\x64\Debug\pagerank_in_memory.exe (进程 8224) 已退出, 代码为 0。
按任意键关闭此窗口, ...

```

图 4.3: 程序运行结果

算法迭代至 69 次时完成收敛, 总用时 32.22 秒, 算法所得排序结果如下:



节点ID	pagerank值
4037	0.004989268865
2625	0.004070534678
6634	0.003726165935
15	0.003098320443
2398	0.002814926042
2328	0.002575268208
2470	0.002543550162
3089	0.002470692576
6946	0.002373668764
3352	0.002363092046
5412	0.002345289244
4191	0.002292234447
7632	0.00226691988
7553	0.002180997453
737	0.00214382738
1297	0.002136097339
3456	0.002110083075
2237	0.002106214345
5254	0.002078229357
6832	0.002070817034
2066	0.001999589657
4712	0.001903874205
762	0.001881513062
7092	0.001878313399

图 4.4: 排序结果

在写入磁盘的情况下, 所得的 top100 节点的排序情况与存储至内存情况下相同, 但所用的迭代次数与所得的 pagerank 值有细微差异, 其原因可能为在将数据写入磁盘时舍弃了部分小数精度。

## 5 Block-based 更新算法

在基本的 Pagerank 算法中, 虽然我们实现了在磁盘中存储了占空间最大的稀疏矩阵, 但是我们发现我们依旧在内存中存储了完整的  $r_{new}$  向量, 该向量的大小为  $1 \times n$  ( $n$  为所有点的数量)。在复杂的图结构中,  $n$  的大小可能超过万级。如果内存无法存储如此大的  $r_{new}$  向量呢? 在这一部分, 我们尝试分块更新  $r_{new}$  向量, 以重复多次加载稀疏矩阵为代价, 进一步减小内存开销。

### 5.1 每次仅计算一部分的 $r_{new}$

在基础分块算法中, 我们简单的将  $r_{new}$  向量按序拆分为多个向量, 但仍使用基础 Pagerank 算法, 即依旧遍历 from 节点, 当我们遍历到 from 节点时, 会挑选处于当前  $r_{new}$  向量范围内的 to 节点更新其  $r_{new}$  值。比如, 0 节点作为 from 节点连接了 0, 1, 3, 5 节点, 而此时的  $r_{new}$  范围为 0-1。则我们仅更新 0, 1 节点的  $r_{new}$  值, 却不更新 3, 5 节点的  $r_{new}$  值。这就意味着我们需要在一次迭代中多次完整遍历稀疏矩阵, 以此为代价使得  $r_{new}$  不用同时均被加载在内存中。

#### 基础分块矩阵

```

1 // 设定r_new向量的分块范围, bsize为分块大小
2 vector<pair<int, int>> block_ranges;
3 for (int start = 0; start < nodes_num; start += bsize) {
4     block_ranges.push_back(make_pair(start, min(start + bsize, nodes_num)));
5 }

```

```

6  ... ..
7  for (int iter = 0; iter < max_iter; ++iter) {
8      // 在每次迭代中依次计算每个r_new向量分块
9      for (const auto& block_range : block_ranges) {
10         ... ..
11         // 判断to点中是否有在当前block_range中的点
12         for (int i = 0; i < degree; ++i) {
13             int to;
14             ss >> to;
15             int idx = id2idx[to];
16             if (block_range.first <= idx && idx < block_range.second) {
17                 // 如果有则更新值，且更新S的值
18                 r_new[idx] += beta * r_ / degree;
19                 r_new_sum += beta * r_ / degree;
20             }
21         }
22     }
23 }

```

## 5.2 $\frac{1-S}{N}$ 值处理

分块算法带来了另外一个问题，即我们无法在第一个分块  $r_{new}$  向量计算完毕时，得出本轮迭代的  $\frac{1-S}{N}$  值，而这部分值应该被加入进每一个  $r_{new}$  中。于是我们对算法略作修改，每次存入  $r_{new}$  向量的值均不包含  $\frac{1-S}{N}$  值，只有当它在下一次迭代中被使用时，我们才会增加这一部分值，如此一来便减少了在一次迭代后，再次读文件写文件的繁琐。

## 5.3 运行结果

在游走参数  $\beta$  取 0.85，收敛条件为 epsilon 取  $1e-6$  时，程序运行结果如下：

```

Microsoft Visual Studio 调试控制台
err 2.92167e-06
finish iter 36
err 2.47475e-06
finish iter 37
err 2.21922e-06
finish iter 38
err 1.94078e-06
finish iter 39
err 1.72254e-06
finish iter 40
err 1.50745e-06
finish iter 41
err 1.32103e-06
finish iter 42
err 1.21181e-06
finish iter 43
err 1.11829e-06
finish iter 44
err 1.03227e-06
finish iter 45
err 9.58249e-07
finish at iter 45
total time: 90.571s.

C:\Users\戴廷钧\source\repos\Pagerank\x64\Debug\Pagerank.exe (进程 23296) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。

```

图 5.5: 程序运行结果

算法迭代至 45 次时完成收敛，用时 90.571 秒。这一时间较普通算法大大增加，主要原因是多次重复读取了完整的稀疏矩阵。算法所得排序结果如下：

4037	0.00493486
2625	0.00401612
6634	0.00367175
15	0.00304391
2398	0.00276051
2328	0.00252085
2470	0.00248913
3089	0.00241628
6946	0.00231925
3352	0.00230868
5412	0.00229087
4191	0.00223782
7632	0.0022125
7553	0.00212658
737	0.00208941
1297	0.00208168
3456	0.00205567
2237	0.0020518
5254	0.00202381
6832	0.0020164
2066	0.00194517
4712	0.00184946
762	0.0018271

图 5.6: 排序结果

所得的 top100 节点的排序情况与普通 Pagerank 算法相比没有区别，而值有细微差异，其原因可能是在文件读写时舍弃了部分小数精度。

## 6 Block-Stripe 更新算法

在上述过程中，我们实现了 Block-based 更新算法，进一步减小了内存的开销，但同时，因为在迭代过程中涉及对邻接矩阵的多次扫描。导致算法的时间性能较低。在此，我们进一步实现 Block-Stripe

更新算法, Block-Stripe 更新算法通过将邻接矩阵进行分块, 使得在更新每一部分  $r^{new}$  向量时, 只需要读取邻接矩阵中与当前的部分  $r^{new}$  相关的一个区域, 而不需要将全部的邻接矩阵进行加载。

## 6.1 邻接矩阵分块

原始的邻接矩阵仍然采用 map 容器结合 NodeData 结构体形式进行实现。邻接矩阵构建完成后, 根据设定的 block\_size 对其进行分块, 每一块邻接矩阵存储至一个单独的磁盘文件中。关键代码如下:

### 矩阵分块

```

1  int blocks_num = nodes_num / block_size + 1;
2  //确定各个块之间的节点范围, 保存在block_ranges中
3  for (int start = 0; start < nodes_num; start += block_size)
4  {
5      block_ranges.push_back(make_pair(start, min(start + block_size, nodes_num)));
6  }
7  for (int i = 0; i < block_ranges.size(); i++)
8  {
9      //对于每一个块都生成其独有的邻接矩阵文件
10     int begin = block_ranges[i].first;
11     int end = block_ranges[i].second;
12     //需要遍历map, 找出所有对当前块中的节点有指向关系的节点。
13     map<int, NodeData> Matrix_for_block;
14     for (auto line : m)
15     {
16         int from = line.first;
17         int degree = line.second.degree;
18         bool scan = false;
19         for (auto dest : line.second.tos)
20         {
21             if (id2Index[dest] >= begin && id2Index[dest] < end)
22             {
23                 scan = true;
24                 Matrix_for_block[from].tos.push_back(dest);
25             }
26         }
27         if (scan)
28             Matrix_for_block[from].degree = degree;
29     }
30     //输出 Matrix_for_block
31     stringstream ss;
32     ss << i;
33     string outfile = matrix_file + ss.str() + ".txt";
34     ofstream out(outfile);
35     for (auto each : Matrix_for_block)
36     {
37         out << each.first << " ";
38         out << each.second.degree << " ";
39         for (int j = 0; j < each.second.tos.size(); j++)

```

```

40         {
41             out << each.second.tos[j] << " ";
42         }
43         out << "\n";
44     }
45 }

```

在生成邻接矩阵的同时也构建起了节点 ID 与其在 vector 数组中的真实索引之间的映射。以 block\_size=1000 为例，此时第一个矩阵区域中，包含的 from 节点都具有相同的特征，即指向索引为 0-999 的节点。并保存至磁盘文件 data\_sparse0.txt 中。

## 6.2 $r^{old}$ 向量的保存

与 Block-based 更新算法面临相同的问题，即每次迭代中的  $\frac{1-S}{N}$  项需要在全部的分块邻接矩阵全部计算完成后才能确定。因此无法及时的对 pagerank 值进行修正。假定初始状态的  $r^{old}$  向量保存在文件 r1 中，在实现 Block-stripe 更新算法的过程中。采用一种中间结果文件 Middle.txt 来保存 pagerank 值计算的中间结果。在确定修正项后，再重新读取中间结果文件，形成最终的  $r^{new}$  向量，并保存在文件 r2 中，在开始下一次迭代之前，交换文件 r1 与文件 r2 的文件名引用。从而使得在下一次迭代中，文件 r2 被视作存储了  $r^{old}$  向量的文件。而文件 r1 用于存储新的  $r^{new}$  向量。在多次迭代中，r1 与 r2 不断的进行交换，保证每次迭代可以获取到正确的  $r^{old}$  向量。迭代过程中的关键代码如下：

### 邻接矩阵读取

```

1     vector<double> curr_block(block_ranges[j].second - block_ranges[j].first , 0);
2     //在每次迭代中，遍历每一个矩阵块
3     stringstream ss;
4     ss << j;
5     string matrix_file = sparse_matrix_file + ss.str() + ".txt";
6     //读取邻接矩阵
7     ifstream martix_In(matrix_file);
8     ifstream old_In(r1);
9
10    string eachline;
11    string oldline;
12
13    int r_old_index = 0;
14    while (getline(martix_In, eachline))
15    {
16        stringstream ss(eachline);
17        int from, degree;
18        ss >> from >> degree;
19        int from_index = id2Index[from];
20
21        while (r_old_index != from_index)
22        {
23            getline(old_In, oldline);
24            r_old_index++;
25        }

```

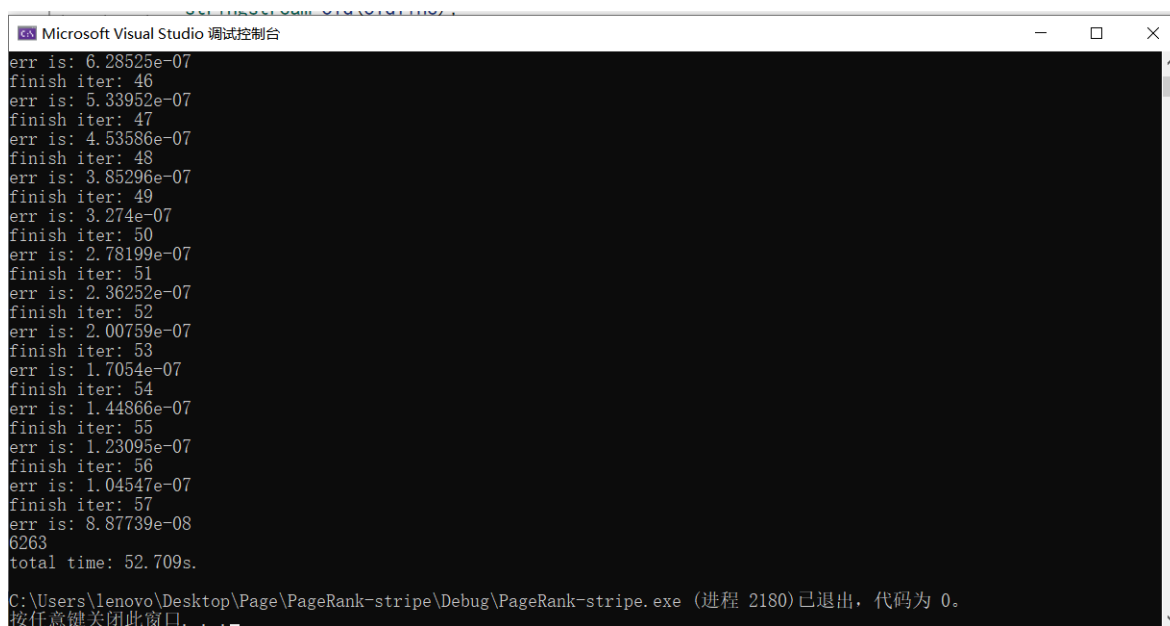
```

26     getline(old_In, oldline);
27     r_old_index++;
28
29     double old_score;
30     stringstream r_ss(oldline);
31     r_ss.precision(std::numeric_limits<double>::max_digits10);
32     r_ss >> old_score;
33     double contribute = beta*old_score / degree;
34     //from节点: from,degree,old_score,以及其指向的若干节点
35     int dest;
36     while (ss >> dest)
37     {
38         int dest_index = id2Index[dest];
39         curr_block[dest_index - block_ranges[j].first] += contribute;
40     }
41 }
42 martix_In.close();
43 old_In.close();

```

### 6.3 运行结果

在游走参数  $\beta$  取 0.85, 收敛条件 epsilon 取  $1e-8$  的条件下, 程序运行结果如下:



```

Microsoft Visual Studio 调试控制台
err is: 6.28525e-07
finish iter: 46
err is: 5.33952e-07
finish iter: 47
err is: 4.53586e-07
finish iter: 48
err is: 3.85296e-07
finish iter: 49
err is: 3.274e-07
finish iter: 50
err is: 2.78199e-07
finish iter: 51
err is: 2.36252e-07
finish iter: 52
err is: 2.00759e-07
finish iter: 53
err is: 1.7054e-07
finish iter: 54
err is: 1.44866e-07
finish iter: 55
err is: 1.23095e-07
finish iter: 56
err is: 1.04547e-07
finish iter: 57
err is: 8.87739e-08
6263
total time: 52.709s.
C:\Users\lenovo\Desktop\PageRank-stripe\Debug\PageRank-stripe.exe (进程 2180) 已退出, 代码为 0。
按任意键关闭此窗口. . .

```

图 6.7: 程序运行结果

算法迭代至 57 次时完成收敛, 总用时 52.709 秒, 算法所得排序结果如下:

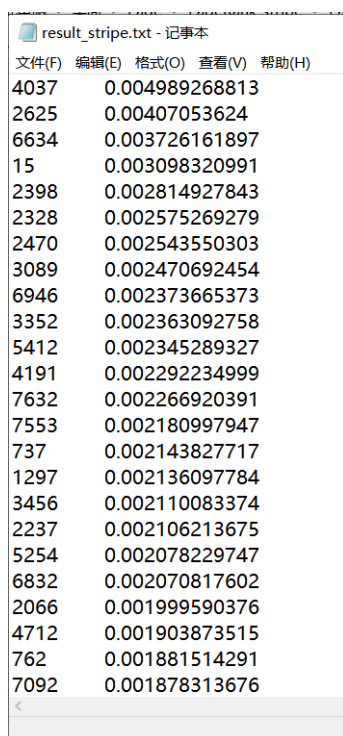


图 6.8: 排序结果

Block-Stripe 更新算法所得的排序结果与基础 pagerank 算法所得结果相近。

## 7 算法性能与结果分析

### 7.1 各算法时间与内存性能比较

在游走参数  $\beta = 0.85$ , 收敛条件  $\epsilon = 1e-6$ ,  $block\_size = 1000$  的条件下, 对各算法运行所需要的内存情况与时间情况进行统计, 对于时间的测量, 采用 C++ 标准库中的 `clock` 函数进行。对于内存测量, 使用 Visual Studio 平台下的性能探查器进行, 各算法运行的时间与内存消耗如下:

算法	迭代次数	运行时间 (s)	内存消耗
PageRank(内存)	43	3.128	4M
PageRank(磁盘)	43	19.059	3M
Block-base	45	116.703	3M
Block-Stripe	43	40.436	2M

由上述表格可得, 在各算法中, 基础 PageRank 算法 (存储至内存) 其用时最短, 同时因为其将所有的数据都存储在内存中, 其内存消耗也是最大的, 在算法运行的全过程中, 其内存消耗始终维持在 5M, 在基础 PageRank 算法 (存储至磁盘) 的改进下, 其内存消耗有一定程度的下降。但运行时间也有所上升。

在 Block-base 更新算法和 Block-Stripe 算法下, 其均是将部分数据存储至内存中, 因此其内存消耗均低于基础的 PageRank 算法。但因其增加了大量的 IO 操作, 因此算法收敛所需的时间远大于基础 PageRank 算法。其中 Block-Stripe 算法与 Block-base 相比, 得益于对邻接矩阵的分条操作, 其需要读取的数据量小于 Block-base 更新算法。体现在算法的运行时间上, 时间性能有近 3 倍的提升。



## 7.2 游走参数对算法的影响

以基础 PageRank(存储至内存) 算法为例, 调整算法中的参数  $\beta$ , 比较算法的收敛情况。分别取  $\beta$  为 1, 0.90, 0.85, 0.75, 0.50, 0.25, 算法的收敛情况如下所示, 其中纵坐标取  $\log_{10}err$ ,

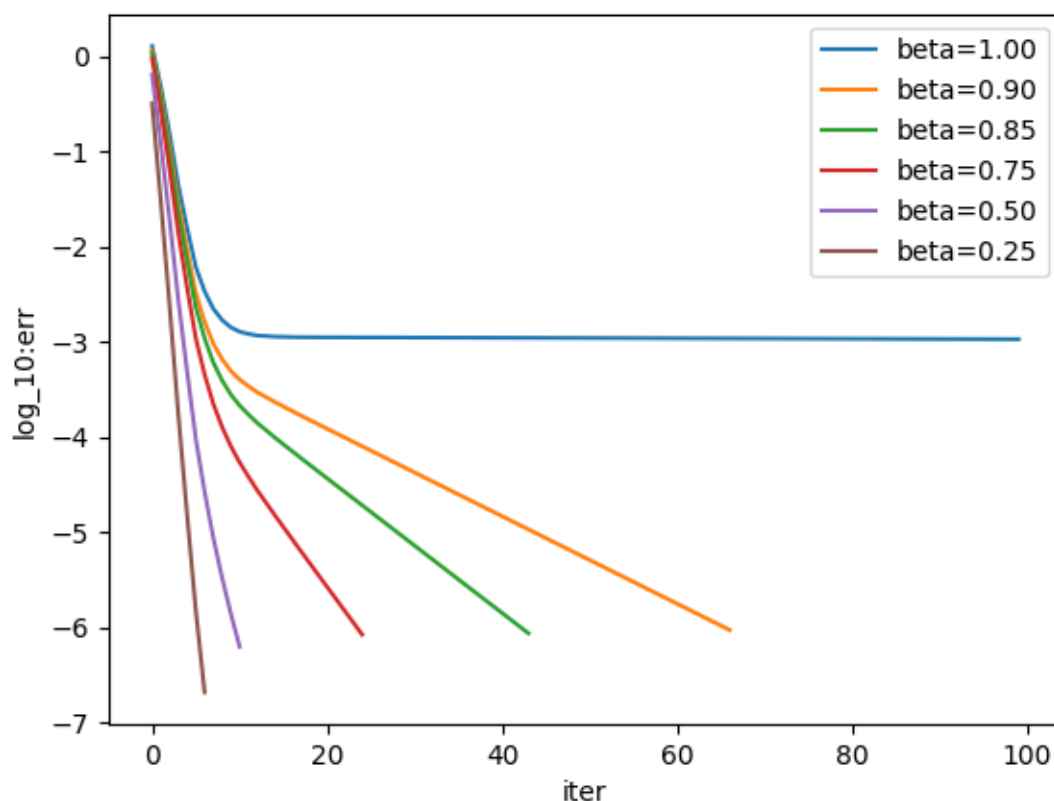


图 7.9: 不同参数下算法收敛情况

由图中情况可以得出, 游走参数  $\beta$  对算法的收敛速度有显著影响。 $\beta$  值越大, 算法的收敛速度越慢。其中当  $\beta$  值为 1 时, 此时相当于不进行随机游走, 算法陷入到 spider-trap 中, 最终无法收敛。在不同的参数下, 排序所得的最终结果也不尽相同。

## 7.3 分块大小对算法性能的影响

以 Block-Stripe 更新算法为例, 分别调整 block\_size 为 100, 200, 400, 600, 800, 1000, 2000, 记录算法运行时间

block_size	time(s)
100	113.223
200	77.718
400	57.188
600	49.936
800	44.797
1000	43.152
2000	37.387