



南开大学
Nankai University

南 开 大 学

计算机学院 & 网络空间安全学院

PageRank 算法设计、优化与改进

成员 1: 苗发生 2010224 信息安全 & 法学双学位

成员 2: 徐文斌 2010234 计算机卓越班

成员 3: 李俞萱 2013572 计算机科学与技术 3 班

指导教师: 杨征路

2023 年 4 月 25 日

目录

一、 前言	1
二、 项目背景	1
(一) PageRank 算法	1
1. 两个重要假设	1
2. 公式定义	2
(二) PageRank 算法应用场景	2
三、 项目说明	2
(一) 环境配置	2
(二) 文件结构	3
(三) 数据集说明	3
四、 程序实现	3
(一) Makefile 文件	3
(二) 封装 PageRank 类	4
(三) 读取数据并分块处理	5
(四) Update 算法	7
(五) PageRank 算法核心	8
(六) 并行加速	9
(七) 程序执行	10
五、 实验结果与分析	11
(一) 基本输出	11
(二) 实验一：块数对算法性能的影响	12
1. 方案与流程	12
2. 结果与分析	12
(三) 实验二：线程数对算法性能的影响	13
1. 方案与流程	13
2. 结果与分析	13
(四) 实验三：阻尼系数对算法结果的影响	15
1. 方案与流程	15
2. 结果与分析	15
六、 总结	17
七、 参考文献	18
A 附录：阻尼系数为 0.85 时的实验结果	18

一、前言

本小组在实现基本作业要求和学有余力的基础上，完成了 PageRank 算法的实现、改进与优化，如表6所示

表 1: 作业要求及本组完成情况

序号	作业要求	是否要求	本组完成情况
1	基础 PageRank 代码	√	√
2	解决 Dead Ends 和 Spider Trap 问题	√	√
3	优化稀疏矩阵	√	√
4	实现分块计算	√	√
5	并行加速		√

二、项目背景

(一) PageRank 算法

PageRank 算法 基本想法是在有向图上定义一个随机游走模型，即一阶马尔可夫链，描述随机游走者沿着有向图随机访问各个结点的行为。在一定条件下，极限情况访问每个结点的概率收敛到平稳分布，这时各个结点的平稳概率值就是其 PageRank 值，表示结点的重要度。PageRank 是递归定义的，PageRank 的计算可以通过迭代算法进行。

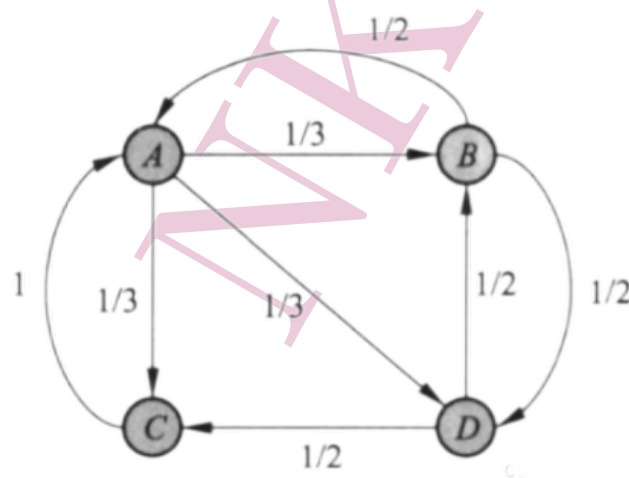


图 1: 有向图

入链数: 指向该节点的链接数

出链数: 由该节点指出的链接数

以上图为例: A 的入链数为 2, 出链数为 3, 所以将由 A 指向其他节点的边权重设置为 $1/3$, 表示 A 访问 B、C、D 节点的概率均为 $1/3$

1. 两个重要假设

数量假设: 在 Web 图模型中, 如果一个页面节点接收到的其他网页指向的入链数量越多, 那么这个页面越重要。

质量假设: 指向页面 A 的入链质量不同, 质量高的页面会通过链接向其他页面传递更多的权重。所以越是质量高的页面指向页面 A, 则页面 A 越重要。

2. 公式定义

当一个网页被其他网页链接得越多且质量越高时, 它的 Pagerank 值就越高。同时, 网页的 Pagerank 值还会受到其链接出去的其他网页的影响。如果一个网页链接出去的网页质量越高, 那么它的 Pagerank 值就会受到更大的影响。而阻尼因子则是用来控制整个网页图的收敛速度, 即防止网页图出现环路而导致计算结果不收敛。公式定义如下:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

其中, $PR(p_i)$ 表示网页 p_i 的 Pagerank 值, N 表示网页总数, $M(p_i)$ 表示所有链接到网页 p_i 的网页集合, $L(p_j)$ 表示网页 p_j 的出链数量, d 是一个介于 0 和 1 之间的常数, 通常取值为 0.85, 称为阻尼因子。

(二) PageRank 算法应用场景

PageRank 算法是一种用于评估网页重要性的算法, 它可以被应用于很多场景中, 其中一些典型的应用场景包括:

- **搜索引擎排名** 搜索引擎使用 PageRank 算法来确定哪些网页应该出现在搜索结果的前面
- **社交网络分析** 在社交网络分析中, PageRank 算法被用来识别哪些用户具有最大影响力
- **推荐系统** 推荐系统可以使用 PageRank 算法来为用户推荐最有可能感兴趣的内容, 例如电影、书籍等
- **网络安全** PageRank 算法可用于检测网络中的恶意节点和欺诈行为
- **生物信息学** 在生物信息学中, PageRank 算法可以被用来确定蛋白质相互作用网络中的关键节点

总的来说, PageRank 算法可以被用于任何需要对网络中节点的重要性进行评估的场景中。

三、项目说明

(一) 环境配置

本次 PageRank 算法设计、优化与改进均在 Linux 系统上使用 C++ 语言完成, 具体环境配置如下:

- **Ubuntu** version 20.04.4
- **g++** version 9.4.0
- **gcc** version 9.4.0
- **gdb** version 9.2

由于用到了并行技术, 因此我们还将用到 `lpthread` 编译选项

(二) 文件结构

在终端输入 `tree` 命令，得到项目结构如图2所示

```
shirly@shirly-virtual-machine:~/bigdata$ tree
.
├── Data.txt
├── main.cpp
├── Makefile
├── PageRank.cpp
└── PageRank.h

0 directories, 5 files
```

图 2: 文件结构

在本项目的 5 个文件中，`Data.txt` 为数据集，`Makefile` 用于自动化编译，其余文件中则为实现 PageRank 算法的代码。我们在 `PageRank.h` 中定义了关键数据类型 `PageRank`，在 `main.cpp` 中明确了程序执行流程，在 `PageRank.cpp` 中实现了关键函数。

(三) 数据集说明

本次实验的数据集为文本文件 `Data.txt`，它描述了一张有向图中结点之间的连接关系。文件共 83852 行数据，每行描述了一条有向边，其格式为

`< FromNodeID > < ToNodeID >`

`FromNodeID` 为头结点的 ID，`ToNodeID` 为尾结点的 ID

四、 程序实现

为了获得较快的运行速度，且能够在较短时间内处理较大的数据集，我们选择 C++17 作为我们本次项目的主要语言，下面，我们将从项目工程代码和实现功能的角度出发，介绍我们的项目实施。

(一) Makefile 文件

为了使得程序能够更方便的运行和展示，我们编写了 `Makefile` 文件，具体实现如下：

Makefile

```
1 cc = g++
2 prom = pagerank
3 deps = $(shell find ./ -name "*.h")
4 src = $(shell find ./ -name "*.cpp")
5 obj = $(src:%.cpp=%.o)
6
7 FLAG = -lpthread -std=c++17
8
9 $(prom): $(obj)
10     @ $(cc) -o $(prom) $(obj) $(FLAG)
11
```

```

12 %.o: %.cpp $(deps)
13     @ $(cc) -c $< -o $@ $(FLAG)
14
15 run: $(prom)
16     @ ./$(prom)
17
18 clean:
19     @ rm -rf $(obj) $(prom)

```

使用说明:

1. 通过 make run 我们可以直接运行程序
2. 通过 make clean 我们可以清除程序运行所产生的中间代码、存储文件和可执行文件
3. 通过 g++ -o pagerank main.cpp PageRank.cpp -lpthread -std=c++17 可以对代码进行编译

(二) 封装 PageRank 类

为了使得程序具有较高的可扩展性，我们采用面向对象的思想，提前封装 PageRank 类，通过实现该类来完成我们的项目。我们的 PageRank 类设计如下：

PageRank 类

```

1  class PageRank
2  {
3  private:
4      std::string input_file;
5      std::string output_file;
6      double beta;
7      int iteration;
8      int block_num;
9      int topn;
10     int thread_num;
11
12     /* data structure */
13     std::unordered_map<int, std::unordered_set<int>> graph;
14     std::unordered_map<int, rank_type> node_rank;
15     int nr_node;
16     /**
17      * 处理数据
18      */
19     void process_data();
20     /**
21      * 迭代更新rank值
22      */
23     void iter(std::unordered_map<int, rank_type> &);
24     /**
25      * page rank过程
26      */

```

```

27 void page_rank();
28 /**
29  * 保存结果
30  */
31 void save_result();
32
33 public:
34     PageRank(double beta, int block_num, int iteration,
35             int topn = 100, int thread_num = 1,
36             std::string input_file = "Data.txt",
37             std::string output_file = "Result.txt");
38     void process();
39 };

```

代码解释：

- 我们定义私有函数 void process_data() 来实现数据的读取，该函数支持分块存储读取的数据。
- 我们定义私有函数 void iter(std::unordered_map<int, rank_type> &) 用来迭代更新节点 PageRank 值
- 我们定义私有函数 void page_rank() 来执行 PageRank 的具体过程
- 我们定义 void save_result() 来保存执行过程的最终结果，并将其存储到 txt 文件中

说明：由于 `unordered_map<>` 的底层实现为 Hash 表，查询的时间开销为 $O(1)$ ，而 `map` 的底层实现为红黑树，查询的时间开销为 $O(\log_2 n)$ 。为了提高读取和修改图中节点 PageRank 值的速度，我们利用 `unordered_map<int, std::unordered_set<int>>` 数据结构来对图进行存储，利用 `std::unordered_map<int, rank_type> node_rank` 来对图中节点的 PageRank 值进行存储。

(三) 读取数据并分块处理

本次实验的数据集为文本文件 Data.txt，共 83852 行数据，每行的格式为 `< FromNodeID > < ToNodeID >`，我们需要将数据读入 `unordered_map<>` 中。

为了保证大数据也可以加载到内存来，我们采取分块处理，分块处理的核心代码如下所示：

读取数据

```

1 void PageRank::process_data()
2 {
3     if (block_num == 1)
4     {
5         ..... 不用分块的时候 .....
6     }
7     else
8     {
9         int max_node_num = 0;
10        int block_num = this->block_num;
11
12        input.open(input_file);

```

```

13     while (!input.eof())
14     {
15         input >> from_node >> to_node;
16         max_node_num = std::max(std::max(from_node, to_node),
17                                 max_node_num);
18     }
19     input.close();
20
21     int step = max_node_num / block_num;
22
23     auto belong_file = [&](int node_num)
24     {
25         int file_num = node_num / step + 1;
26         file_num = std::min(file_num, block_num);
27         return file_num;
28     };
29
30     std::ofstream output;
31     for (int i = 1; i <= block_num; i++)
32     {
33         graph.clear();
34         input.open(input_file);
35         while (!input.eof())
36         {
37             input >> from_node >> to_node;
38             if (belong_file(from_node) == i)
39                 graph[from_node].insert(to_node);
40             if (belong_file(to_node) == i)
41                 graph[to_node].insert(from_node);
42         }
43         nr_node += graph.size();
44         output.open("block" + std::to_string(i));
45         for (auto &edge : graph)
46         {
47             output << edge.first;
48             for (auto &to_node : edge.second)
49             {
50                 output << " " << to_node;
51             }
52             output << "\n";
53         }
54         input.close();
55         output.close();
56         for (auto &[from_node, to_nodes] : graph)
57         {
58             node_rank[from_node] = 1.0 / nr_node;
59             node2mutex[from_node] = true;
60         }
61     }
62 }

```



```

60     }
61 }
62 }

```

代码解释：由于不分块时数据读取较为简单，我们省略这部分，只对分块部分的代码进行介绍。在分块处理的时候，我们先利用 `max_node_num = std::max(std::max(from_node, to_node), max_node_num)` 得到图中节点的最大值，即图中节点的数量。然后依据块数可以算得每个块存储的点数，接着按照 `<from_node,to_node>` 的格式，将图分块存储到对应的块文件中，并将该块写入磁盘。每当我们写完一块时，我们需要根据入链数和出链数计算对应节点的 PageRank 值。

(四) Update 算法

在 Update 算法中，我们遵循：

$$r^{(t+1)} = M \cdot r^t$$

但在算法实现上，我们仍然是按照迭代更新的形式（因为邻接关系不再是以邻接矩阵的方式组织，而是以列表的形式）。另外，为使得更新操作更快，我们采用并行的方式进行加速更新过程。具体实现过程如下所示：

并行加速的 Update 算法

```

1 void PageRank::iter(std::unordered_map<int, rank_type> &next_rank)
2 {
3     if (thread_num == 1)
4     {
5         for (auto &[from_node, to_nodes] : graph)
6         {
7             int out_degree = to_nodes.size();
8             for (auto to_node : to_nodes)
9             {
10                 next_rank[to_node] += beta * (node_rank[from_node] /
11                     out_degree);
12             }
13         }
14     }
15     else if (thread_num > 0)
16     {
17         std::vector<std::thread*> threads;
18         threads.resize(thread_num);
19         for (int thread_id = 0; thread_id < thread_num; thread_id++)
20         {
21             threads[thread_id] = new std::thread(update_rank, std::ref(graph)
22                 , std::ref(node_rank), std::ref(next_rank), thread_id,
23                 thread_num, beta);
24         }
25         for (int thread_id = 0; thread_id < thread_num; thread_id++)
26         {
27             threads[thread_id] -> join();
28         }
29     }
30 }

```

```

25         delete threads[thread_id];
26     }
27 }
28 else
29 {
30     std::cout << "thread num illegal" << std::endl;
31     exit(-1);
32 }
33 }

```

代码解释：当线程数量为 1 时，我们进行单线程的处理；当线程数量大于 1 时，我们动态创建所有线程，并行化加速更新过程。另外，我们还进行安全性检查，当线程数量小于 1 时，我们输出 thread num illegal，程序结束。

注意，当线程运行结束时，我们需要用 delete 将线程清除，避免占用过多的内存空间以及存在安全隐患。

（五） PageRank 算法核心

PageRank 是本项目的核心,我们通过不断调用 iter 函数(iter 函数是我们自己实现的 Update 函数) 来实现，具体实现如下：

PageRank 算法核心

```

1 void PageRank::page_rank()
2 {
3     /* 初始时每个节点的rank值为1/N, 这里block_num不为1的时候在process_data中
4        处理 */
5     if (block_num == 1)
6         for (auto &[from_node, to_nodes] : graph)
7         {
8             node_rank[from_node] = 1.0 / nr_node;
9             node2mutex[from_node];
10        }
11    std::unordered_map<int, rank_type> next_rank;
12    for (int i = 0; i < iteration; i++)
13    {
14        /* next_ran清零 */
15        for (auto &[node, rank] : node_rank)
16            next_rank[node] = 0;
17
18        if (block_num == 1)
19        {
20            /* 更新rank */
21            iter(next_rank);
22        }
23        else
24        {
25            int from_node, to_node;
26            std::ifstream input;

```

```

26         std::string line;
27         std::string node;
28         int num = 0;
29         for (int file_num = 1; file_num <= block_num; file_num++)
30         {
31             graph.clear();
32             input.open("block" + std::to_string(file_num));
33             while (getline(input, line))
34             {
35                 std::istringstream ss(line);
36                 ss >> from_node;
37                 graph[from_node];
38                 while (ss >> to_node)
39                 {
40                     graph[from_node].insert(to_node);
41                 }
42             }
43             input.close();
44             iter(next_rank);
45         }
46     }
47     /* 远程传播 */
48     rank_type sum = 0, inc = 0;
49     for (auto &[node, rank] : next_rank)
50         sum += rank;
51     inc = (1 - sum) / nr_node;
52     for (auto &[node, rank] : next_rank)
53         rank += inc;
54     node_rank = next_rank;
55     std::cout << "iteration: " << i + 1 << " over" << std::endl;
56 }
57 }

```

代码解释：PageRank 是本项目最核心的函数，我们首先需要初始化每个节点的 PageRank 值 (PR)，初始化为 $\frac{1}{N}$ ，之后便是迭代更新的过程，迭代次数为我们预先设定的次数，在每一次的迭代过程中，我们需要调用更新函数，更新结束后，进行远程传播，并输出一定的提示词 (iteration i over, 即第几轮更新完毕)。执行完毕后，程序执行结束，并输出性能指标 (如总耗时等)

(六) 并行加速

并行加速以使得项目能够支持超大数据集的处理是我们的一大优势，为了使得我们的程序能够处理较大的数据集，我们采用并行化的方式加速程序的运行，并行化实现的核心代码如下：

并行化加速

```

1  /* 计算每个线程处理的矩阵的范围 */
2  int step = graph.size() / all_threads;
3  int begin = thread_id * step;
4  int end = ((thread_id == all_threads - 1) ? graph.size() : begin + step);

```

```

5  auto begin_it = graph.begin();
6  std::advance(begin_it, begin);
7  auto end_it = graph.begin();
8  std::advance(end_it, end);
9  /* 并行化更新和处理节点的PageRank值 */
10 std::vector<std::thread*> threads;
11 threads.resize(thread_num);
12 for (int thread_id = 0; thread_id < thread_num; thread_id++)
13 {
14     threads[thread_id] = new std::thread(update_rank, std::ref(graph), std::
        ref(node_rank), std::ref(next_rank), thread_id, thread_num, beta);
15 }
16 for (int thread_id = 0; thread_id < thread_num; thread_id++)
17 {
18     threads[thread_id]-->join();
19     delete threads[thread_id];
20 }

```

代码解释：在并行加速的过程中，为避免不同线程之间的冲突，我们需要首先计算不同线程所处理矩阵的范围（我们采用平均分的策略，即所有线程要处理的量相同），并得到线程所对应矩阵的初始和结束位置。之后为每一个线程动态分配其内存空间，在更新迭代结束之后，我们再通过 delete 清理掉所有的线程。

（七） 程序执行

为了使程序具有更高的可扩展性，我们利用面向对象的思想对程序进行设计。因此，在程序执行之时，我们只需要初始化 PageRank 类，即可完成我们本项目的需求。

程序执行和初始化

```

1  int main()
2  {
3      PageRank *prk = new PageRank(0.85, 4, 100, 100, 8);
4      prk->process();
5      delete prk;
6  }
7
8  PageRank::PageRank(double beta, int block_num, int iteration,
9                      int topn, int thread_num,
10                      std::string input_file,
11                      std::string output_file)
12      : beta(beta), block_num(block_num), iteration(iteration),
13        topn(topn), thread_num(thread_num),
14        input_file(input_file), output_file(output_file)
15  {
16      nr_node = 0;
17  }

```

代码解释：在本次实验中，我们所用的参数如下：

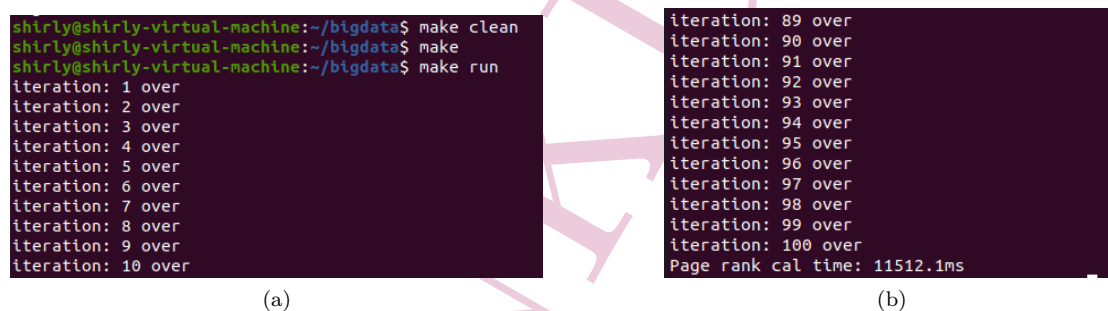
1. 阻尼系数: 0.85
2. 块数: 4
3. 迭代次数: 100
4. 输出 top 节点的个数: 100
5. 线程数量: 8

注意: 我们留了非常方便的接口, 如果需要对以上参数进行修改, 直接修改 PageRank 初始化函数的参数即可! 在实验结果与分析阶段, 我们也通过控制变量法, 不断调整参数进行对比分析和测试。

五、 实验结果与分析

(一) 基本输出

令阻尼系数为 0.85、分块数为 4、迭代次数为 100、输出条目数为 100、线程数为 8, 我们在终端依次输入 make clean, make, make run, 得到终端输出如图3所示



```
shirly@shirly-virtual-machine:~/bigdata$ make clean
shirly@shirly-virtual-machine:~/bigdata$ make
shirly@shirly-virtual-machine:~/bigdata$ make run
iteration: 1 over
iteration: 2 over
iteration: 3 over
iteration: 4 over
iteration: 5 over
iteration: 6 over
iteration: 7 over
iteration: 8 over
iteration: 9 over
iteration: 10 over
iteration: 89 over
iteration: 90 over
iteration: 91 over
iteration: 92 over
iteration: 93 over
iteration: 94 over
iteration: 95 over
iteration: 96 over
iteration: 97 over
iteration: 98 over
iteration: 99 over
iteration: 100 over
Page rank cal time: 11512.1ms
```

(a)

(b)

图 3: 终端输出

每一轮迭代完成时终端都有“iteration: xx over”的输出, 且迭代完成后会输出总耗时。此外观察到目录下生成了结果文件 Result.txt, 其内容如图4, 里面按照 [NodeID] [Score] 的格式记录了每个节点的分数

```

1 4037 0.004550721329
2 2625 0.003838895787
3 6634 0.003793950524
4 15 0.003150047683
5 2398 0.002670001353
6 2328 0.002612516732
7 5412 0.002380151036
8 2470 0.00237847265
9 7632 0.002280095617
10 3089 0.002257379116
11 3352 0.002227904094
12 737 0.002181135759
13 4191 0.00214805973
14 3456 0.002139477817
15 2237 0.002130234649
16 5254 0.002117760083
17 6832 0.002082023284
18 7553 0.00207630818
19 2066 0.002018862091
20 1297 0.001975515512
21 7092 0.001897691915
22 4310 0.00187456438
23 4712 0.001821311123
24 6946 0.001786863434
25 6774 0.001784267397
26 762 0.001770609479
27 1186 0.001750290386
28 993 0.001698821611
29 2958 0.001694177577
30 6006 0.001653635539
31 2657 0.001638825236
32 4335 0.001637876975
33 4828 0.001633926032
34 665 0.001630763795
35 4735 0.001629321046
36 8042 0.001617639665
37 4875 0.001611446438
38 4256 0.001555841145
39 3537 0.001553581375
40 7620 0.001542997623
41 3238 0.001524164046
42 4264 0.001406630660

```

图 4: Result.txt

阻尼系数为 0.85 的具体实验结果参见附录 1。

(二) 实验一：块数对算法性能的影响

1. 方案与流程

为直观验证算法运行时间会随着块数的增加线性增长，我们在其他条件完全一致（阻尼系数为 0.85、迭代次数为 100、线程数为 8）的情况下，记录分块数为 1~8 时实际算法的运行时间。

2. 结果与分析

经过实验，我们得到运行时间随分块数变化的数据如表2所示

表 2: 实验一结果数据

块数	时间 (s)
1	9.68364
2	30.2106
3	30.9922
4	35.5844
5	40.0065
6	43.2788
7	43.9803
8	48.663

为清晰看出运行时间随着分块数的变化，我们绘制折线图如图5所示

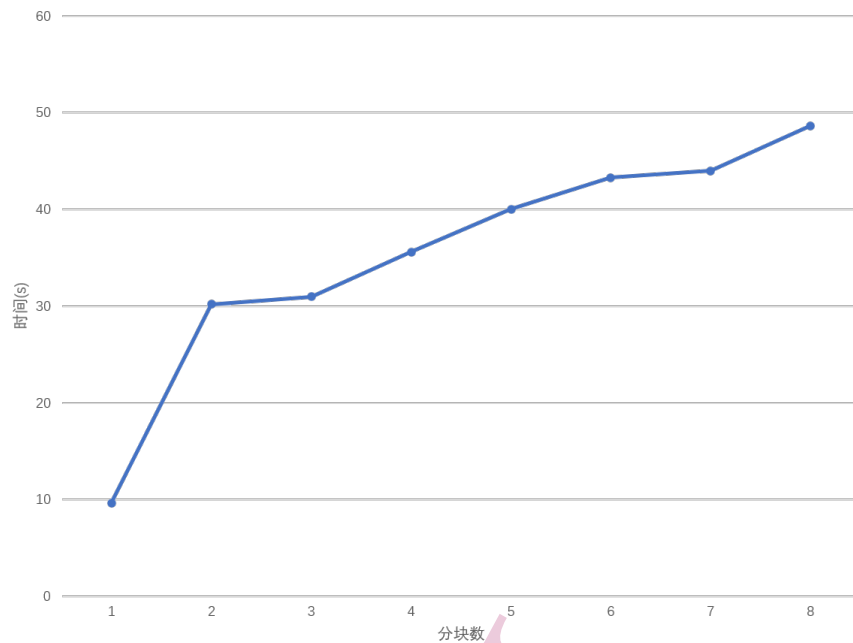


图 5: 运行时间随分块数的变化

从图中可知，运行时间随着分块数的增加近乎线性增长，这是由于分块时 IO 带来了额外的时间消耗。从理论上而言，令迭代数量贡献的复杂度为 $O(max_iter)$ ，由于每一分块都会运行 max_iter 的迭代次数，故总体时间损耗大致为

$$O(max_iter \times Blocks \times Complexity_of_PageRank)$$

因此，当迭代次数一定，PageRank 基本算法一定的情况下，总复杂度应当随着分块数增加而几近线性增加。实验结果与理论预期一致。

(三) 实验二：线程数对算法性能的影响

1. 方案与流程

为验证多线程能够加快算法运行时间，提高算法效率，我们在其他条件完全一致（阻尼系数为 0.85、分块数为 1、分迭代次数为 100）的情况下，记录线程数为 1~8 时实际算法的运行时间。

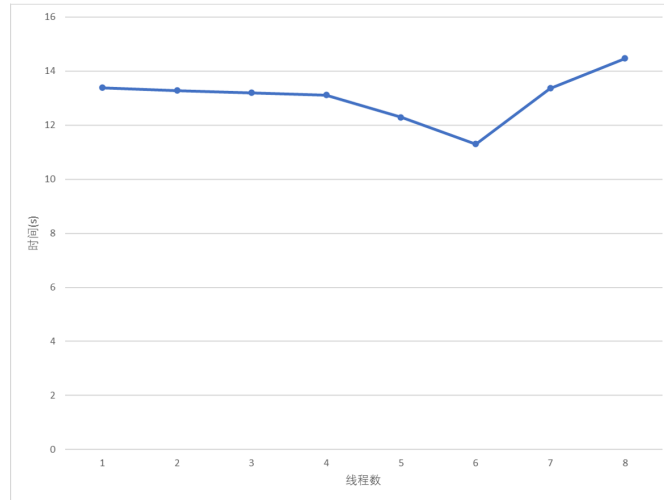
2. 结果与分析

我们得到实验结果数据如表3所示

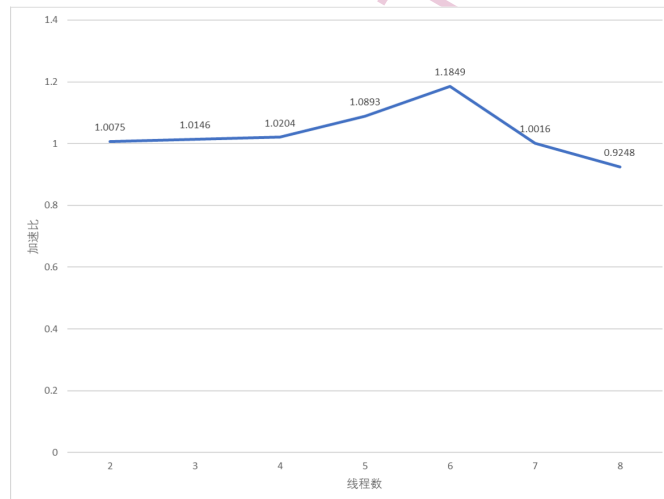
表 3: 实验二结果数据

线程数	时间 (s)
1	13.385
2	13.2852
3	13.1928
4	13.1176
5	12.288
6	11.2964
7	13.3637
8	14.4733

为清晰看出运行时间随着线程数的变化，我们绘制折线图如图6(a) 所示；为进一步看出并行加速效果，我们计算加速比如图6(b) 所示



(a) 运行时间随线程数的变化



(b) 加速比随线程数的变化

图 6: 运行结果与线程数

从图表中可知

- 从运行时间的角度，在当前规模的数据集下，线程数为 6 时具有最少的运行时间，最佳的性能。理论上而言，当线程数目增加时，越来越多的线程并行工作能够带来更大的并行优化效果进而减少总体运行时间，但同时也伴随着越来越多的线程通信等额外开销，影响运行效率。当线程数小于等于 6 时，运行时间随着线程数的增加而减小；当线程数大于 6 时，多线程的额外开销逐渐多于并行能带来的性能提升，因此运行时间随着线程数的增加而增加。
- 从加速比的角度，在当前规模的数据集下，线程数为 6 时加速比最大。当线程数小于等于 6 时，加速比的增长随着线程数的增加而逐渐加快。多线程并行技术的引入，从时间上能给 PageRank 算法带来将近 20% 的性能提升

当数据集规模进一步增大的时候，多线程并行带来的性能提升效果将更加显著。

(四) 实验三：阻尼系数对算法结果的影响

1. 方案与流程

为探究阻尼系数对算法结果（结点的 pagerank 计算结果）的影响，我们在分块数为 1、线程数为 8 的情况下，记录不同阻尼系数：0.65, 0.75, 0.85, 0.95 下 TOP100 结点的分值。阻尼系数的大小会影响收敛速度，因此我们参考《Google's PageRank and Beyond: The Science of Search Engine Rankings》一书中阻尼系数与收敛所需的迭代次数的关系（如表4所示），根据阻尼系数的大小调整迭代次数，保证程序结束时已经收敛。

表 4: 阻尼系数与建议迭代次数

阻尼系数	迭代次数
0.5	34
0.75	81
0.8	104
0.85	142
0.9	219
0.95	449
0.99	2292
0.999	23015

2. 结果与分析

由于实验数据过多，故不再列表展示。我们将不同阻尼系数下计算出的 TOP100 结点的分值绘制成折线图如图7所示，再计算 100 个结点的分值标准差并列表如5所示。在图表中，我们能清晰看到阻尼系数不同时分值大小的差异，以及分值密集程度的差异

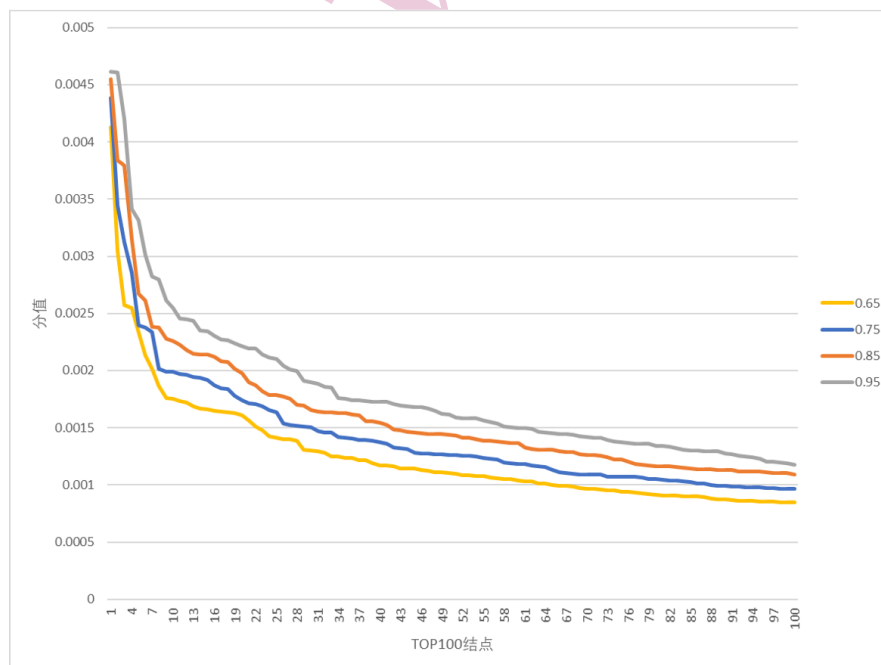


图 7: 不同阻尼系数下 TOP100 结点的分值

表 5: 阻尼系数与标准差

阻尼系数	TOP100 分值标准差
0.65	0.000503626
0.75	0.000551164
0.85	0.000598263
0.95	0.000667315

从图表中可知，**阻尼系数越大，TOP100 结点的分值普遍更大，且标准差也更大，即不同结点的 pagerank 值差异越大**。从理论上来说，结点 A 的 pagerank 值满足如下公式，公式中 d 为阻尼系数，PR 表示 pagerank：

$$PR(A) = (1 - d) + d(PR(T_1)/C(T_1) + \dots + PR(T_n)/C(T_n))$$

从公式中得知：阻尼系数越小，不同节点的 pagerank 值差异越小，即节点的 pagerank 的计算结果越密集；阻尼系数越大，不同节点的 pagerank 值整体差异越大，即节点的 pagerank 的计算结果越均匀。实验结果与理论预期一致。

六、 总结

完成 Pagerank 算法的实验，是一次深入学习和探究该算法的机会。Pagerank 算法是一种基于图论的算法，用于评估网络中节点的重要性。该算法具有重要的理论和实践价值，在搜索引擎优化、社交网络分析等领域得到了广泛应用。

本次项目中，小组成员（苗发生、徐文斌和李俞萱）均在代码、实验和报告中有所贡献，我们对 PageRank 算法较为感兴趣，一起进行了深入地学习和讨论，并对算法基础进行实现、改进与优化，我们有如下感想：

- **深入理解 PageRank 算法原理：**深入理解算法是实现程序的第一步，在算法实现的过程中，我们小组花费了大量时间学习算法的原理和基本概念，包括随机浏览模型、马尔可夫链、迭代计算和收敛性等方面。这些知识让我们对算法的理解更加深入和全面。

- **实践操作提高编程能力：**考虑到程序的性能等影响，我们选择了较为底层的 C++ 而非 python 作为我们本次实现的编程语言。在实现的过程中，我们没有调用已经存在的包和头文件，所有的功能和实现都通过手搓来完成，这极大的提高了我们的编程能力

- **探究算法的局限性和改进方法：**在实验中，我们发现 Pagerank 算法存在一些局限性，例如对于大规模网络的处理效率不高，容易受到垃圾节点的影响等。通过不断尝试和改进，我们尝试了基于随机游走、并行计算等技术手段，使得算法在应对大规模网络和垃圾节点方面更加有效。算法运行效率显著提高

- **思考算法在实际应用中的价值和挑战：**通过完成本项目，我们深入思考了 Pagerank 算法在搜索引擎、社交网络分析、推荐系统、网络安全和生物信息学等领域的应用和挑战，包括算法的可靠性、公平性、隐私保护等方面。这些思考让我们更加清晰地认识到算法的实际应用价值和未来发展方向。

总的来说，PageRank 算法的设计、实现与改进让我们从理论到实践，深入学习和掌握了该算法的细节和应用。这不仅提高了我们的学术研究能力，也为我们今后在相关领域的研究和工作奠定了坚实的基础。

七、参考文献

1. Broder A Z, Lempel R, Maghoul F, et al. Efficient PageRank approximation via graph aggregation[C] Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters. 2004: 484-485.
2. Maslov S, Redner S. Promise and pitfalls of extending Google's PageRank algorithm to citation networks[J]. Journal of Neuroscience, 2008, 28(44): 11103-11105.
3. Tao M, Yang X, Gu G, et al. Paper recommend based on LDA and PageRank[C]//Artificial Intelligence and Security: 6th International Conference, ICAIS 2020, Hohhot, China, July 17-20, 2020, Proceedings, Part III 6. Springer Singapore, 2020: 571-584.
4. Ding Y. Topic-based PageRank on author cocitation networks[J]. Journal of the American Society for Information Science and Technology, 2011, 62(3): 449-466.
5. Amy N. Langville, Carl D. Meyer. Google's PageRank and Beyond: The Science of Search Engine Rankings[M]. USA: Princeton University Press, 2014. 47-66.
6. 徐键. 基于 PageRank 的科技论文推荐系统 [J]. 电子世界, 2013 (1): 104-105.
7. 李江, 孙建军. 链接分析对引文分析的启示: 从 PageRank 到 Paperank[J]. 情报学报, 2009 (4): 618-625.
8. 徐家树, 邢立新, 覃征. 超链接文本相关度的 PageRank 算法 [J]. 哈尔滨工业大学学报, 2009, 41(1): 223-225.
9. 张付志, 石占伟, 郭学敏. 一种抗击链接垃圾页面的 PageRank 改进算法 [J]. 信息安全与通信保密, 2009 (8): 77-79.
10. 陈祖琴, 张惠玲, 葛继科, 等. 基于加权关联规则挖掘的相关文献推荐 [J]. 现代图书情报技术, 2007 (10): 57-61.

A 附录：阻尼系数为 0.85 时的实验结果

在阻尼系数为 0.85、分块数为 4、迭代次数为 100、输出条目数为 100、线程数为 8 的情况下, 实验结果如表6所示

表 6: 实验结果

序号	结点 ID	pagerank	序号	结点 ID	pagerank
1	4037	0.004550721	51	2485	0.001429684
2	2625	0.003838896	52	5079	0.001415331
3	6634	0.003793951	53	6784	0.001410852
4	15	0.003150048	54	2871	0.001397812
5	2398	0.002670001	55	2535	0.001386438
6	2328	0.002612517	56	2654	0.001385833
7	5412	0.002380151	57	5404	0.001378284
8	2470	0.002378473	58	6334	0.001370007
9	7632	0.002280096	59	8163	0.001368607
10	3089	0.002257379	60	5459	0.001368436
11	3352	0.002227904	61	2565	0.001323964
12	737	0.002181136	62	4400	0.001313241
13	4191	0.00214806	63	5543	0.001309919
14	3456	0.002139478	64	3562	0.001306718
15	2237	0.002138235	65	4600	0.00130474
16	5254	0.00211776	66	2859	0.001295128
17	6832	0.002082823	67	6059	0.001287811
18	7553	0.002076308	68	3334	0.001285826
19	2066	0.002018862	69	2746	0.001271041
20	1297	0.001975516	70	2651	0.001264214
21	7092	0.001897692	71	7809	0.00125856
22	4310	0.001874564	72	5022	0.001252469
23	4712	0.001821311	73	3897	0.001241962
24	6946	0.001786863	74	5226	0.001223241
25	6774	0.001784267	75	3321	0.001221327
26	762	0.001770609	76	3034	0.001200329
27	1186	0.00175029	77	2576	0.001184813
28	993	0.001698822	78	1633	0.001176588
29	2958	0.001694178	79	1211	0.001167483
30	6006	0.001653636	80	4040	0.001164779
31	2657	0.001638825	81	1726	0.001164091
32	4335	0.001637877	82	4531	0.001163814
33	4828	0.001633926	83	7961	0.001157211
34	665	0.001630764	84	5563	0.00114958
35	4735	0.001629321	85	4981	0.001146037
36	8042	0.00161764	86	6330	0.001138878
37	4875	0.001611446	87	1842	0.001138628
38	4256	0.001555841	88	3005	0.001133779
39	3537	0.001553581	89	86	0.001131642
40	7620	0.001542998	90	1754	0.00113093
41	3238	0.001524164	91	5605	0.001126799
42	4261	0.00148468	92	3962	0.001119151
43	271	0.001474784	93	2594	0.001117332
44	5123	0.001462679	94	7890	0.001117122
45	3568	0.001460283	95	3443	0.001113883
46	5484	0.001454841	96	4666	0.001110016
47	1549	0.001445221	97	7214	0.001106296
48	3498	0.001443898	98	3260	0.001104792
49	825	0.001442289	99	28	0.001104332
50	3084	0.001441677	100	6124	0.001087686