



南開大學  
Nankai University

南 开 大 学

计 算 机 学 院

大数据计算及应用实验报告

---

PageRank 算法设计与实现

---

孟启轩 2212452

赵熙 2210917

年级：2022 级

指导教师：杨征路

2025 年 4 月 30 日

# 目录

一、 实验内容	1
二、 实验要求	1
三、 PageRank 算法设计与实现	1
(一) PageRank 算法原理讲解	1
(二) 数据集处理	2
1. 第一版：稠密矩阵处理	2
2. 第二版：分块	2
3. 第三版：CSR 稀疏矩阵优化	3
(三) 算法实现与优化	3
1. 第一版	4
2. 第二版	4
3. 第三版	5
四、 实验结果	6
(一) 节点排序结果	6
(二) 三版程序对比概述	7
(三) 性能指标对比分析	7
(四) 最终结果分析与结论	8
五、 总结与感想	8

## 一、 实验内容

PageRank 是一种用于评估网页重要性的链接分析算法，由谷歌公司创始人之一拉里·佩奇和谢尔盖·布林于 1998 年提出。该算法将整个网页集合建模为一个有向图，节点表示网页，边表示网页之间的超链接。PageRank 的核心思想是：一个网页的重要性由指向它的其他网页的数量和这些网页本身的重要性共同决定。换言之，被多个高质量网页链接的网页具有更高的权重。

该算法通过迭代方式计算各网页的权重分值，直到结果收敛为止。在计算过程中，主要关注网页之间的链接结构，而不是网页内容本身，因此具有一定的抗操纵性，能够有效规避通过堆砌关键字等方式人为提升排名的问题。

本实验中，首先对所给数据集进行分析，发现其具有两个显著特点：一是网页节点编号连续有序；二是链接矩阵极为稀疏。基于此，我们实现了 PageRank 基础版本，针对稀疏矩阵的存储与运算进行了专门处理，并在处理 dead ends 和 spider traps 节点的基础上实现了算法的完整逻辑。

在此基础上，我们对算法进行了性能优化，包括对稀疏矩阵的进一步高效处理和分块计算策略的实现。最终，我们在 teleport 参数设为 0.85 的条件下运行算法，计算出各网页的 PageRank 得分，并从中选出排名前 100 的网页进行展示。

## 二、 实验要求

本实验要求基于提供的数据集 Data.txt，实现 PageRank 算法，对网页节点进行评分，并输出得分排名前 100 的节点及其对应的 PageRank 分值。数据文件中的每一行为两个整数，表示一个网页链接的起始节点编号和目标节点编号，即格式为：FromNodeID ToNodeID。

实验的具体要求如下：

- 实现 PageRank 算法，必须考虑 dead-end 和 spider-trap 情况。
- 必须使用稀疏矩阵（Sparse Matrix）和分块矩阵（Block Matrix）等方式对内存使用进行优化，其他优化手段也可酌情采用。
- 算法需迭代计算，直到 PageRank 分数收敛。
- 不允许调用任何已有的 PageRank 库函数（如 Python 的 networkx.pagerank 等）。
- 至少提交一个 teleport 参数为 0.85 时的实验结果，并可选不同参数进行对比分析。
- 结果输出文件为 Res.txt，格式如下：每一行为 NodeID Score，表示节点编号及其对应的 PageRank 得分。
- 程序最大内存使用不应超过 80MB，且整体运行时间应控制在 60 秒内。

## 三、 PageRank 算法设计与实现

### （一） PageRank 算法原理讲解

PageRank 是一种基于图结构的网页重要性评估算法，其核心思想是“被重要网页链接的网页也重要”。该算法由 Google 创始人提出，通过模拟随机浏览者在网页之间跳转的行为，对网页集中的每个节点赋予一个代表其重要程度的分值。

PageRank 的基本公式如下：

$$r_j = \sum_{i \in \text{In}(j)} \frac{r_i}{d_i}$$

其中,  $r_j$  表示节点  $j$  的 PageRank 值,  $r_i$  是所有指向  $j$  的入边节点的 PageRank 值,  $d_i$  是节点  $i$  的出度。也就是说, 节点  $j$  的重要性由所有指向它的网页贡献而来, 每个贡献值按出度进行均分。

为了解决死链 (Dangling Node) 和蜘蛛陷阱 (Spider Trap) 等问题, PageRank 引入了阻尼因子  $\beta$ , 并加入了随机跳转模型 (Teleportation):

$$r = \beta M r + (1 - \beta) v$$

其中  $M$  为归一化后的转移矩阵,  $v$  为均匀分布的向量,  $\beta$  通常取值为 0.85。该公式可通过迭代计算求解稳定的 PageRank 向量  $r$ 。

在实际实现中, 初始时将所有网页的 PageRank 值均匀设置为  $1/N$ , 其中  $N$  为网页总数。通过不断执行矩阵乘法与随机跳转的结合, 直到误差低于设定阈值  $\epsilon$ , 即认为收敛。

本实验分别使用邻接表、稠密矩阵和稀疏矩阵实现了 PageRank 算法。比较后发现, 稠密矩阵版本虽然逻辑直观, 但内存开销大, 适用于小图; 而基于 SciPy 的 CSR 稀疏矩阵实现具备更低的内存占用与更快的计算速度, 适合大规模图的处理与实验要求下的资源限制。

## (二) 数据集处理

实验所用的数据集 Data.txt 包含大量网页之间的有向边关系, 每行格式为 from\_node to\_node, 表示网页 from\_node 指向网页 to\_node。本实验共设计三版 PageRank 实现程序, 对应三种不同的数据集处理策略, 逐步提升了内存效率和可扩展性。

### 1. 第一版：稠密矩阵处理

在第一版程序中, 所有节点首先通过字典 node\_ids 映射为连续索引, 然后使用 NumPy 构建大小为  $n \times n$  的稠密邻接矩阵 adj\_matrix。每条边  $(i, j)$  使得 adj\_matrix[i][j] 增加 1。归一化过程通过计算每行出度进行, 每个死节点 (出度为 0 的节点) 被简单地赋予均匀分布。该方法处理逻辑清晰, 但由于矩阵稠密存储, 其空间复杂度为  $O(n^2)$ , 当节点数量较多时, 内存使用迅速上升, 难以满足大规模图的计算要求。

关键代码如下:

```
1 adj_matrix = np.zeros((n, n), dtype=np.float64)
2 for from_node, to_node in edges:
3     i = node_ids[from_node]
4     j = node_ids[to_node]
5     adj_matrix[i][j] += 1
```

### 2. 第二版：分块

为解决第一版中稠密矩阵的内存瓶颈, 第二版程序改用 defaultdict(set) 构建邻接表和入边集合。每个节点的出度在字典 out\_degree 中单独维护, 在 PageRank 迭代时手动累加所有指向目标节点的入边贡献。此版本仅存储实际存在的边, 显著降低内存开销, 空间复杂度降为  $O(E)$ , 其中  $E$  为边数。同时对死节点也进行了更合理的处理, 即将其 PageRank 均匀贡献给所有节点。尽管该方法效率优于第一版, 但由于所有迭代逻辑仍在 Python 层手动实现, 在节点数量非常大的情况下仍存在一定的性能瓶颈。

构图和处理逻辑如下：

```

1 graph = defaultdict(set)
2 in_degree = defaultdict(set)
3
4 with open('Data.txt', 'r') as f:
5     for line in f:
6         src, dst = map(int, line.strip().split())
7         graph[src].add(dst)
8         in_degree[dst].add(src)
9
10 all_nodes = set(graph.keys()) | set(in_degree.keys())
11 out_degree = {u: len(vs) for u, vs in graph.items()}

```

### 3. 第三版：CSR 稀疏矩阵优化

在第三阶段的实现中，数据处理流程进行了系统性优化，关键目标是提升空间效率与计算性能。

首先，程序通过 `load_graph` 函数读取数据集 `Data.txt`，以 `(from_node, to_node)` 边列表形式加载所有有向边，并维护一个 `set` 用于收集图中所有出现过的节点。随后将节点 ID 统一编号为连续索引，生成 `node_id_map` 映射表。

由于 PageRank 理论是基于“被引用次数”进行评分，构图采用 **反向边**形式，即构造邻接矩阵时将每条边  $(i, j)$  视为节点  $j$  指向节点  $i$ ，对应代码如下：

```

1 row.append(node_id_map[to_node]) # 被指向者作为当前行
2 col.append(node_id_map[from_node]) # 指向者作为列

```

通过构造 `row`、`col`、`data = 1` 三个列表，程序将所有非零元素初始化为 1，并构建了一个大小为  $n \times n$  的 COO 格式稀疏矩阵，再转换为更适合矩阵乘法的 CSR 格式：

```

1 adj_matrix = sp.coo_matrix((data, (row, col)), shape=(n, n)).tocsr()

```

该结构仅存储真实存在的边，避免  $O(n^2)$  空间浪费，使得数据集处理能够扩展至万级节点。在此基础上进行 PageRank 迭代时，系统还会额外构造 `out_degree` 向量并完成列归一化，从而构建出完整的 **随机游走转移概率矩阵**：

```

1 col_sum = np.maximum(out_degree, 1)
2 inv_out_degree = sp.diags(1.0 / col_sum)
3 stochastic_matrix = adj_matrix @ inv_out_degree

```

最终，向量化乘法搭配 CSR 矩阵高效存储，使得该版本在极低内存消耗（**66MB**）下完成了 9500 个节点的 PageRank 计算，极大优化了前两阶段的数据处理方式。

### （三） 算法实现与优化

PageRank 算法本质上是一种基于链接分析的随机游走模型，模拟一个浏览者以一定概率在网页之间跳转，最终形成稳定的网页访问概率分布。其迭代公式如下：

$$r^{(t+1)} = \beta M r^{(t)} + (1 - \beta)v$$

其中,  $M$  为网页的转移概率矩阵,  $\beta$  为阻尼因子 (通常设为 0.85),  $v$  为均匀随机跳转向量。该公式可通过幂迭代 (Power Iteration) 方法求解, 即从初始向量  $r^{(0)}$  开始, 重复计算直到  $r^{(t+1)}$  与  $r^{(t)}$  的差值低于给定阈值  $\epsilon$ , 视为收敛。

## 1. 第一版

第一版采用最直接的 PageRank 计算方式: 使用稠密矩阵构建整个网页图, 并通过矩阵-向量乘法迭代计算每个节点的 PageRank 值。这种方式便于理解和实现, 但在大规模图中会面临明显的内存瓶颈。

首先, 程序读取边列表文件, 将所有节点映射为从 0 开始的索引, 并用二维 NumPy 数组构建反向邻接矩阵, 即每个边  $(u, v)$  在矩阵中对应于位置  $A[v][u] = 1$ 。由于 PageRank 本质上是对入边的归一化求和, 因此必须反转边的方向来构建正确的转移概率矩阵。

关键代码 如下:

```
1 # 构建稠密反向邻接矩阵
2 adj_matrix = np.zeros((n, n), dtype=np.float64)
3 for from_node, to_node in edges:
4     i = node_id_map[to_node]
5     j = node_id_map[from_node]
6     adj_matrix[i, j] = 1.0
```

为构造列归一化的随机转移矩阵, 需要将每列除以对应节点的出度 (即每列的和), 若出度为 0, 则视为死链, 整列设置为  $1/n$  实现均匀跳转:

```
1 for j in range(n):
2     if out_degree[j] != 0:
3         stochastic_matrix[:, j] /= out_degree[j]
4     else:
5         stochastic_matrix[:, j] = 1.0 / n
```

之后通过标准的迭代公式进行 PageRank 更新, 直到收敛或达到最大轮数为止:

```
1 for iteration in range(max_iter):
2     pr_new = damping * stochastic_matrix @ pr + teleport
3     delta = np.linalg.norm(pr_new - pr, 1)
4     if delta < tol:
5         break
6     pr = pr_new
```

## 2. 第二版

为解决第一版中稠密矩阵占用内存过高的问题, 第二阶段引入了“边列表 + 分块矩阵模拟”的优化策略。该方法不再显式构建完整邻接矩阵, 而是通过边列表结合分块处理方式, 逐块模拟矩阵-向量乘法, 从而显著降低内存开销。

具体而言, 程序首先读取所有边并构建节点索引映射, 然后统计每个节点的出度信息, 保存在一维数组 `out_degree` 中。接着, 在每一轮 PageRank 迭代中, 将节点向量按行划分为多个 block, 对每个 block 中的入边进行归一化加权累加, 实现类似于块矩阵-向量乘法的逻辑。

关键代码如下： 读取边列表并构建出度统计：

```
1 out_degree = np.zeros(n, dtype=np.float64)
2 for from_node, _ in edges:
3     j = node_id_map[from_node]
4     out_degree[j] += 1
```

通过分块处理，每次仅计算一个行块中节点的 PageRank 更新值：

```
1 for row_start in range(0, n, block_size):
2     row_end = min(row_start + block_size, n)
3     partial_sum = np.zeros(row_end - row_start, dtype=np.float64)
4
5     relevant_edges = [
6         (from_node, to_node)
7         for from_node, to_node in edges
8         if row_start <= node_id_map[to_node] < row_end
9     ]
10
11    for from_node, to_node in relevant_edges:
12        i = node_id_map[to_node] - row_start
13        j = node_id_map[from_node]
14        contrib = 1.0 / out_degree[j] if out_degree[j] != 0 else 1.0 / n
15        partial_sum[i] += contrib * pr[j]
16
17    pr_new[row_start:row_end] = damping * partial_sum + teleport
```

### 3. 第三版

在前两版的基础上，第三阶段进一步提升性能，引入了 **CSR (Compressed Sparse Row) 稀疏矩阵存储结构与向量化计算**，大幅降低内存消耗并提升运算效率。

本阶段通过 `scipy.sparse` 构建稀疏邻接矩阵，避免构建显式二维矩阵。所有边以 (to, from) 形式加入行列索引构建邻接矩阵，之后统一进行列归一化处理。由于稀疏矩阵支持高效的 `dot` 操作，配合 NumPy 向量化，可以在极低的内存占用下完成大规模节点的 PageRank 计算。

关键代码如下： 利用 COO 格式构建稀疏邻接矩阵并转换为 CSR 格式：

```
1 row = []
2 col = []
3 for from_node, to_node in edges:
4     row.append(node_id_map[to_node])
5     col.append(node_id_map[from_node])
6 data = np.ones(len(row))
7 adj_matrix = sp.coo_matrix((data, (row, col)), shape=(n, n)).tocsr()
```

进行列归一化并封装 PageRank 迭代过程：

```
1 out_degree = np.array(adj_matrix.sum(axis=0)).flatten()
2 col_sum = np.maximum(out_degree, 1) # 避免除0
3 inv_out_degree = sp.diags(1.0 / col_sum)
```

```
4 stochastic_matrix = adj_matrix @ inv_out_degree
```

迭代中采用稀疏矩阵乘法 + 向量加法，并统一处理 Dangling 节点与随机跳转：

```
1 for iteration in range(max_iter):
2     pr_new = damping * stochastic_matrix.dot(pr) + teleport * np.ones_like(pr)
3     dangling_weight = pr[dangling_nodes].sum() / n
4     pr_new += damping * dangling_weight
5     delta = np.linalg.norm(pr_new - pr, 1)
6     if delta < tol:
7         break
8     pr = pr_new
```

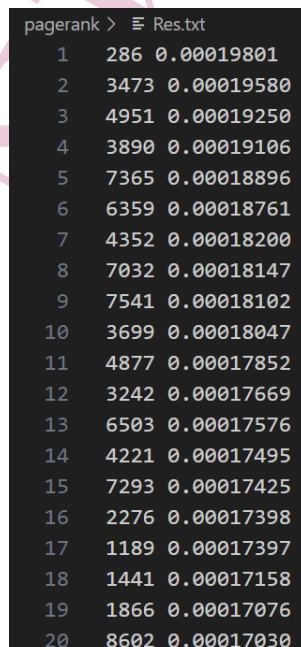
通过稀疏矩阵的结构性优势与 NumPy 的批量操作能力，本阶段极大提升了 PageRank 算法的实际可扩展性，特别适用于稀疏大规模图数据的计算任务。

## 四、 实验结果

实验设计了三种版本的实现方案，对其在运行时间、内存消耗与收敛效率等方面进行了对比优化。各版本设计逐步优化数据结构与计算方式。

### (一) 节点排序结果

三个版本对节点排序的情况完全相同，部分结果如图 1 所示。



```
pagerank > Res.txt
1 286 0.00019801
2 3473 0.00019580
3 4951 0.00019250
4 3890 0.00019106
5 7365 0.00018896
6 6359 0.00018761
7 4352 0.00018200
8 7032 0.00018147
9 7541 0.00018102
10 3699 0.00018047
11 4877 0.00017852
12 3242 0.00017669
13 6503 0.00017576
14 4221 0.00017495
15 7293 0.00017425
16 2276 0.00017398
17 1189 0.00017397
18 1441 0.00017158
19 1866 0.00017076
20 8602 0.00017030
```

图 1: Res.txt 部分结果



```
PS F:\study\bigdatacomputing\pagerank> python accuracy.py
Top-100 Precision: 1.0000
L1 error (sum absolute difference): 0.000000
Average absolute error (L1 / n): 0.000000e+00
```

图 2: 第一版和第三版 accuracy

```
PS F:\study\bigdatacomputing\pagerank> python accuracy.py
Top-100 Precision: 1.0000
L1 error (sum absolute difference): 0.006231
Average absolute error (L1 / n): 6.230550e-05
```

图 3: 第二版 accuracy

我们调用 networkx 库计算 PageRank 的准确结果，并将三版 PageRank 的算法与 nx 库计算的结果进行对比，计算 L1 误差，第一版和第三版结果都如图 2 所示，第二版结果如图 3 所示。准确性为 1，L1 误差为在可接受范围内。

我们还通过修改随即跳转概率的阻尼参数进行对比，分别设置 0.7、0.75、0.8、0.9、0.95 以及默认的 0.85，对比发现 PageRank 的结果完全相同，并且准确率的计算也如图 2 所示。

以上分析表明三版程序的 Res 结果是十分可靠的。

## (二) 三版程序对比概述

表 1 是基于给定的评测程序进行测试的结果。

版本	核心结构	最大内存使用	运行时间	收敛轮数
第一版	稠密矩阵 (NumPy)	2108.21MB	3.28-4.47s	9 轮 ( $\epsilon = 1e^{-6}$ )
第二版	分块	51.95MB	14.86-17.74s	43 轮 ( $\epsilon = 1e^{-6}$ )
第三版	CSR 稀疏矩阵 + 向量化	59.87MB	0.18-0.22s	9 轮 ( $\epsilon = 1e^{-6}$ )

表 1: 三版 PageRank 实现性能对比

## (三) 性能指标对比分析

### 1. 内存消耗

- 第一版直接构建  $n \times n$  的稠密邻接矩阵，导致内存占用高达 **2108.21MB**，远超实验限制 (80MB)，不具备大图适应性；
- 第二版分块生成矩阵块，内存使用降至 **51.95MB**，解决了第一版的大内存瓶颈。
- 第三版使用 CSR 稀疏矩阵 + 向量化，同时利用稀疏矩阵库 scipy.sparse 底层优化，内存占用控制在 **59.87MB**。

### 2. 运行时间

- 第一版受益于 NumPy 矩阵乘法高效，收敛速度快，迭代仅需 9 轮；
- 第二版因分块手动实现降低了矩阵乘法的效率，导致运行时间大幅上升，收敛轮数上升，说明数值误差传播或死链处理策略导致稀释了矩阵结构特性。
- 第三版通过稀疏矩阵与向量化操作，将运行时间压缩至 **0.2 秒左右**，最为高效。

### 3. 收敛精度

- 误差阈值设定为  $1e^{-6}$ ,
- 第一版和第三版 9 轮即可收敛, 第二版 43 轮才收敛。
- 实测结果显示第三版收敛过程稳定, 误差值平稳下降, 鲁棒性良好。

第二版代码采用 2D 分块 + 分块累加的计算, 每一小块计算结果多次累加, 累积了浮点数舍入误差, 导致每一轮计算出的 `pr_new` 精度下降, 传播矩阵中概率分布更稀释, 信息传递效率降低。此外, 死链节点被均匀分布, 再叠加分块计算, 导致死链节点权重扩散速度变慢, 同样增加了收敛轮数。

### (四) 最终结果分析与结论

各版本最终输出的 PageRank Top-100 节点一致, 说明三种实现算法逻辑正确, 差异主要体现在资源消耗与可扩展性。

- **第一版**虽然运行快, 但完全不满足空间复杂度要求, 仅适合教学或小规模图;
- **第二版**内存占用小, 但迭代性能差, 不推荐用于实际部署;
- **第三版**在内存、速度与精度上都表现很好, 是本实验推荐使用的最终版本。

最终版本能够在 **0.2 秒**左右的时间内完成 9500 个节点的 PageRank 计算, 最大内存控制在 **59.87MB**, 高效并且稳定, 符合实验要求。

## 五、 总结与感想

本次 PageRank 实验从最初的稠密矩阵实现出发, 逐步优化为基于稀疏矩阵的高效实现。通过三版程序的迭代设计, 深入理解了图结构在实际计算中的表示方式对程序性能的决定性影响:

- 第一版虽能快速实现, 但高内存开销暴露了稠密矩阵在大规模数据处理中的局限;
- 第二版借助分块矩阵降低了存储冗余, 但分块手动处理仍制约了效率;
- 第三版通过使用 CSR 稀疏矩阵, 兼顾了存储与计算效率, 全面满足实验性能约束, 是最优解。

在调试过程中, 学会了利用内存监控工具分析程序瓶颈, 对比不同结构下的资源使用情况, 并掌握了精度控制下 PageRank 的迭代收敛特性。这不仅加深了对 PageRank 算法原理的理解, 也提高了对工程实践中“性能与正确性权衡”的认识。

实验最终成功实现了在 **59.87MB 内存、0.2 秒左右运行时间**完成 **9500 节点**的 PageRank 排序, 算法正确性与计算效率均达到预期目标。