南開大學
Nankai University

计算机学院

大数据计算及应用

PageRank 链接分析

姓名：赵康明　李颖

学号：2110937　2110939

专业：计算机科学与技术

2024 年 4 月 29 日

目录

1 实验内容

1. 实现基础 Pagerank 算法的编写，考虑 dead end 和 spider trap 节点
2. 通过优化稀疏矩阵的存储方式提升运算效率
3. 通过矩阵的分块运算与并行计算提升运算效率

2 问题重述

给定节点之间的指向关系进行 PageRank 链接分析，计算网页得分，得到网页的重要性排名。

3 实验环境

- 系统：Windows 11
- 处理器：AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz
- 内存容量：16GB
- 开发环境：PyCharm Community Edition 2023.2
- Python 版本：3.11.4

4 数据集说明

本次实验的数据集为 Data.txt，每行的格式为 <srcNode, destNode>，表示图中存在从 srcNode 到 destNode 的一条边。

对数据进行预处理，得到如下结果：

统计量	数值	备注
数据集尺寸	1.38MB	数据集所占存储空间
结点数	8297	数据集中网页总数
边数	135737	网页总链接数
行数	135737	未去重的网页链接数
Dead ends 数	2187	出链数为 0 的网页数
边缘结点数	0	入链数为 0 的网页数
最大入链节点编号	8185	包含最多入链的网页编号
最大入度	32	网页最大被引用次数
最大出链节点编号	4144	包含最多出链的网页编号
最大出度	43	网页最大引用次数

可以看到，数据行数等于有向边数，说明数据不存在冗余，不必去除重复边。图中存在无出度的结点，即存在 Dead ends，因此处理时需要注意死胡同的情况。

数据集大小为 1.38MB，远小于运行环境下的内存容量。为了实验要求，假设数据集规模过大，无法一次性装载入内存中，因此采用矩阵分块算法。由于网页数为 8297，因此邻接矩阵需保存 8297^2 个数据项，占用内存空间约 262.9 MB。而网页间的链接数只有 135737 条，邻接矩阵中非 0 项占比仅为 0.2%，说明邻接矩阵非常稀疏。因此，除了基本算法之外，还可以优化稀疏矩阵的存储方式，进一步节省内存开销。

5 实验参数设置

为保证实验的可复现性，列出实验中部分参数设置情况如下所示。

参数名	取值	备注
E	1e-15	迭代停止更新容差
beta	0.85	随机游走因子
block_size	50	数据分块尺寸

参数 E 为控制迭代更新算法的阈值，当本次迭代与上次迭代的差值小于 E 时，算法收敛，停止迭代。参数 beta 为随机游走因子，控制算法随机游走的概率。参数 block_size 为分块算法中块尺寸，表示的是 dest 的取值范围。

6 实验原理

6.1 PageRank 算法

6.1.1 基本思想

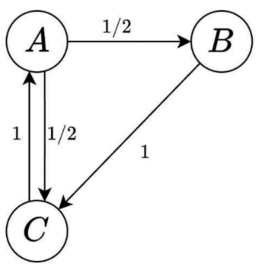
PageRank[1] 是一种衡量互联网网页重要性的算法，其核心思想是基于网络上网页之间的链接关系和用户的随机浏览行为来评估网页的重要程度。该算法假设互联网是一个有向图，其中网页表示节点，超链接表示边。在这个图上，PageRank 通过模拟网页浏览者的随机访问行为，来确定每个网页的权重。

具体来说，PageRank 将互联网视为一个马尔可夫链，即浏览者在网页之间进行随机跳转的过程。在这个过程中，浏览者以等概率随机访问当前网页的超链接，或者以一定的概率跳转到任意一个网页。通过不断迭代计算，最终得到一个平稳分布，即每个网页的 PageRank 值，反映了网页的重要性。

PageRank 的计算涉及到大量的矩阵运算和迭代过程，但其核心思想是简单而直观的：重要的网页会被更多其他网页链接，而这些链接的网页本身也会更具权威性，从而形成一个类似于“链接的链接”网络结构。通过考虑网页之间的链接关系，PageRank 能够有效地评估网页的权重，为搜索引擎提供更加准确和有用的搜索结果。

6.1.2 基本内容

将互联网视为一个有向图，并在此基础上定义随机游走模型，即一阶马尔可夫链。这模型描述了网页浏览者在互联网上随机浏览网页的过程。假设浏览者在每个网页按照连接出去的超链接以等概率跳转到下一个网页，并持续进行这样的随机跳转，从而形成一阶马尔可夫链。PageRank 反映了这个马尔可夫链的平稳分布。每个网页的 PageRank 值即是平稳概率的一种表达。例如下图：



可以将上图视为简略的互联网网络模型，A 链接到 B 和 C，B 只链接到 C，而 C 只链接到 A。在随机游走模型中，从 A 出发的浏览者有 50% 的概率跳转到 B 或 C；从 B 出发的浏览者会 100% 跳转到 C；而从 C 出发的浏览者则会 100% 跳转到 A

基于以上假设模型，我们可以在有向图上定义随机游走模型，即一阶马尔可夫链，其中结点表示状态，有向边表示状态之间的转移，假设从一个结点到通过有向边相连的所有结点的转移概率相等。具体地，转移矩阵是一个一阶矩阵

$$M = [m_{ij}]_{n\times n} \tag{1}$$

注意矩阵有以下性质：

$$m_{ij} \geq 0 \tag{2}$$

$$M = [m_{ij}]_{n\times n} \tag{3}$$

即每个元素非负，每列元素之和为 1 即矩阵 M 为随机矩阵 (stochastic matrix) 。在有向图上的随机游走形成马尔可夫链。也就是说，随机游走者每经一个单位时间转移一个状态，如果当前时刻在第 i 个结点 (状态), 那么下一个时刻在第 j 个结点 (状态) 的概率是 m_{ij} 这一概率只依赖于当前的状态，与过去无关，具有马尔可夫性。

在图 1 的有向图上可以定义随机游走模型。结点 A 到结点 B , C 存在有向边，可以以概率 1/2 从 A 分别转移到 B, C, 并以概率 0 转移到 A, 于是可以写出转移矩阵的第 1 列。结点 B 到结点 C 存在有向边，可以以概率 1 从 B 转移到 C , 并以概率 0 分别转移到 B 和 A, 于是可以写出矩阵的第 2 列，等等。于是得到转移矩阵

$$M = \begin{bmatrix} 0 & 0 & 1 \\ 1/2 & 0 & 0 \\ 1/2 & 1 & 0 \end{bmatrix} \tag{4}$$

随机游走在某个时刻 t 访问各个结点的概率分布就是马尔可夫链在时刻 t 的状态分布，可以用一个 n 维列向量 R_t 表示，那么在时刻 $t+1$ 访问各个结点的概率分布 R_{t+1} 满足

$$R_{t+1} = MR_t \tag{5}$$

给定一个包含 n 个结点 $v_1, v_2, \cdots, v_n$ 的强连通且非周期性的有向图，在有向图上定义随机游走模型，即一阶马尔可夫链。随机游走的特点是从一个结点到有有向边连出的所有结点的转移概率相等，转移矩阵为 M 。这个马尔可夫链具有平稳分布

$$MR = R \tag{6}$$

平稳分布 R 称为这个有向图的 PageRank。R 的各个分量称为各个结点的 $PageRank$ 值。其中 $PR(V_i)$,$i = 1, 2, \cdots, n$, 表示结点的的 PageRank 值。显然有

$$PR\left(v_i\right) \geqslant 0, \quad i = 1, 2, \cdots, n \tag{7}$$

$$PR(v_i) = \sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}, \quad i = 1, 2, \cdots, n \tag{8}$$

这里 $M(v_i)$ 表示指向结点 v_i 的结点集合，$L(v_j)$ 表示结点 v_j 连出的有向边的个数。

PageRank 的基本定义是理想化的情况，在这种情况下，PageRank 存在，而且可以通过不断迭代求得 PageRank 值。

6.1.3 算法问题

Dead End 在实际网络情况中，可能会出现网页节点没有出链的情况，即该网页无指向其他网页的转移。这种情况会导致网页 PageRank 值为 0 的情况。而我们可以通过采用 teleport 的方式预防这种情况，teleport 指的是模拟互联网用户在浏览网页时选择跳转到其他页面的行为。在 PageRank 计算中，当一个用户到达一个没有出度链接的页面，它会以一定的概率随机跳转到任何一个页面，从而避免陷入死胡同。

Spider trap 在 PageRank 算法中，当处理连通图时，如果某个节点只有出链而没有入链，算法会假设该节点能够到达所有其他节点，并对其进行 PageRank 值的计算。然而，如果这个节点处于一个 Spider Trap 中，那么随机游走可能会无限循环，导致 PageRank 值无法收敛。

为了解决这个问题，PageRank 算法引入了阻尼因子（damping factor）。阻尼因子表示用户在浏览网页时停留在当前页面的概率。通过引入阻尼因子，算法可以控制随机游走不会过度陷入 Spider Trap 页面。

具体而言，当算法遇到一个只有出链而没有入链的节点时，如果该节点位于一个 Spider Trap 中，算法会根据阻尼因子的设置减少对该节点的影响。通过减少对 Spider Trap 节点的权值分配，PageRank 算法能够更好地避免陷入无限循环，并确保对其他页面的权值分配更加合理和准确。

修正后得到的公式为：

$$PR(v_i) = d\left(\sum_{v_j \in M(v_i)} \frac{PR(v_j)}{L(v_j)}\right) + \frac{1-d}{n}, \quad i = 1, 2, \cdots, n \qquad (11) \tag{9}$$

- $PR(v_i)$ 表示节点 v_i 的 PageRank 值，即网页 v_i 的重要性。
- d 是阻尼因子 (damping factor), 通常取值在 O 和 1 之间，表示用户从当前网页跳转到其他网页的概率。
- $M(v_i)$ 表示指向节点 v_i 的所有入链集合。
- $L(v_j)$ 表示节点 v_j 的出链数量，即节点 v_j 的出度。
- n 是图中节点的总数。

6.2 BlockStrip 算法

6.2.1 算法概述

BlockStrip 是一种用于优化 Pagerank 计算的方法，它通过对网页图进行分块处理，减少了计算和存储的开销。具体来说，BlockStrip 方法将整个网页图分割成多个较小的子图（block），每个子图包含一定数量的网页节点。然后对每个子图进行 Pagerank 计算，最后将这些子图的 Pagerank 值合并得到

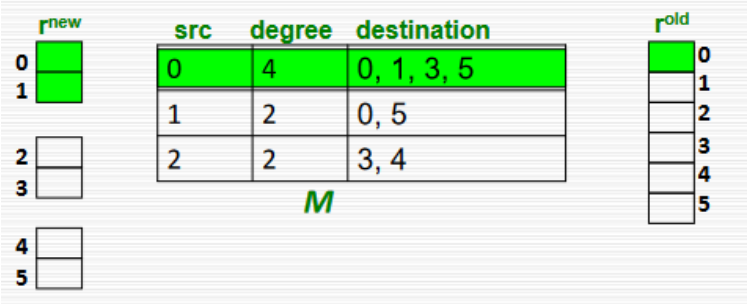
整体的 Pagerank 值。每个子图默认是单台计算机可存储并处理的，整合操作也默认是单台计算机可处理的。这样就能使大规模数据集被多台计算机配合处理，从而得到最终结果。

具体过程如下所示：

1. **划分块**：指定块大小为 block_size，将整个网页集合划分为多个块，每个块包含一部分网页。块的划分可以根据网页的链接关系进行，确保每个块内的网页之间存在链接关系。
2. **初始化**：为每个网页设置一个初始的 PageRank 值，通常设置为 1/N，其中 N 是网页的总数。
3. **局部计算**：依次将划分完的块读入内存，在每个块内部进行局部计算，据 PageRank 的公式更新每个网页的 PageRank。
4. **迭代更新**：整合局部计算的结果后，重复计算步骤，直到所有网页的 pagerank 收敛。

6.2.2 具体实现

分割后的矩阵 M 每一行的格式为 <src, degree, destination>

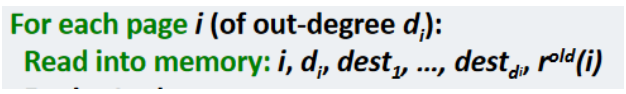


将 r^{new} 分成 k 块，每块的大小为 block_size。每次计算只需读取一块到内存中，同时将稀疏矩阵 M 根据所要更新的目标结点的值分成相应的块，计算时从磁盘中加载所需块，减少计算所需内存。假设计算机内存足以装得下大小为 N 的列向量 r^{new}，则不必将 r^{new} 写入磁盘，只需要一直放在内存中即可，方便计算。

计算 PageRank 值时，首先初始化 r^{new} 为 (1-)/N：

$$r^{new} = (1 -)N \tag{10}$$

依次到磁盘上读取已经建立好的 Block Strip 分块，得到分块索引 i（即 src）、对应 dest 列表 $dest_i$，同时从 r^{old} 中读取 $r^{old}(i)$。



遍历 src 的 dest 列表，一次迭代更新的公式如下所示：

$$r^{new}(dest_j) += * r^{old}(i)/d_i \tag{11}$$

6.2.3 算法问题

对于 Dead ends 和 Spider trap 的情况，可采用与 PageRank 中的同一方法进行处理：每轮更新完毕后将结果写入 r^{temp} 中，最后读取 r^{temp} 中的值，加和得到 S，计算 1-S/N，作为修正写入到 r^{new} 中。

遍历 r^{new}，将值累加到 S 上。

$$S+ = r^{new}(dest) \tag{12}$$

按照如下公式更新 r^{new} 即可。

$$r^{new}(dest_j)+ = (1 - S)/N \tag{13}$$

7 核心代码

7.1 基础 PageRank

7.1.1 数据存储

在实验数据的读取部分，我们创建了一个 Graph 类对象和 load_data 函数，以下对其进行介绍：

Graph 类 创建了两个成员变量 graph 和 all_nodes，分别用于存储图的邻接表和所有节点的集合。因为目标为优化稀疏矩阵，我们可以采取使用邻接表的方式去存储稀疏矩阵，降低内存的使用。graph 是一个字典，键表示节点，值是该节点所指向的所有邻居节点的列表。all_nodes 是一个集合，用于存储图中的所有节点。

```
class Graph:
def __init__(self):
    self.graph = {}
    self.all_nodes = set()

def add_edge(self, from_node, to_node):
    """
    向图中添加一条边。
    """
    self.all_nodes.add(from_node)
    self.all_nodes.add(to_node)
    if from_node not in self.graph:
        self.graph[from_node] = []
    self.graph[from_node].append(to_node)
```

load_data 函数 该函数打开指定路径的数据文件，逐行读取数据。对于每一行，使用 map(int, line.strip().split()) 将字符串转换为整数列表，其中第一个整数表示起始节点，第二个整数表示目标节点。调用 Graph 类的 add_edge 方法，将边添加到图中。

```python
def load_data(file_path):
"""
读取数据，构建图并返回图对象。
"""
graph = Graph()

with open(file_path, 'r') as file:
    for line in file:
        from_node, to_node = map(int, line.strip().split())
        graph.add_edge(from_node, to_node)

return graph
```

7.1.2 PageRank 计算

对于基础 PageRank 的实现，我们需要使用矩阵乘法的方式迭代计算出 PageRank 值。主要步骤为：

1. 构建转移矩阵 S：根据输入的图和节点信息，构建一个初始的转移矩阵 S，其中矩阵的每个元素表示节点之间的链接关系。
2. 计算总转移矩阵 M：根据转移参数、转移矩阵 S 和节点数量，计算总转移矩阵 M，该矩阵包含了转移参数和随机跳转的影响。
3. 初始化 PageRank 值：将所有节点的 PageRank 值初始化为相等的初始值，以便开始迭代计算。
4. 迭代更新 PageRank 值：
   - 通过矩阵乘法迭代计算 PageRank 值，直到收敛或达到最大迭代次数为止。在实现具体的矩阵乘法中，由于手写的乘法的运行速度较慢，是用来 numpy 库中的矩阵乘法进行计算。
   - 在每次迭代中，将当前 PageRank 值与上一次迭代的值进行比较，直到它们的差异小于给定的容差 (tol)。

```python
def calculate_page_rank(graph, teleport_parameter=0.85 ,tol=1e-15,
    max_iterations=1000):
"""
基于给定的图、节点和参数,计算PageRank值。
"""
all_nodes = graph.all_nodes
N = len(all_nodes)
node_idx = {node: i for i, node in enumerate(sorted(all_nodes))}
# 构建转移矩阵M
M = np.zeros([N, N], dtype=np.float64)
# 构建转移矩阵S
```

```
    S = np.zeros([N, N], dtype=np.float64)
    for out_node, in_nodes in graph.graph.items():
        for in_node in in_nodes:
            S[node_idx[in_node], node_idx[out_node]] = 1
    # 处理矩阵
    for col in range(N):
        sum_of_col = S[:, col].sum()
        if sum_of_col == 0:
            S[:, col] = 1 / N
        else:
            S[:, col] /= sum_of_col
    # 计算总转移矩阵M
    E = np.ones((N, N), dtype=np.float64)
    M = teleport_parameter * S + (1 - teleport_parameter) / N * E
    # 初始化PageRank值
    P = np.ones(N, dtype=np.float64) / N
    # 迭代更新PageRank值，直到收敛或达到最大迭代次数
    for iteration in range(max_iterations):
        prev_P = np.copy(P)
        ## P=MP
        ## P=M@p
        P=np.dot(M,P)
        # P = matrix_multiply(M, P)
        diff = np.linalg.norm(P - prev_P)
        if diff < tol:
            print(f'Converged after {iteration + 1} iterations.')
            break
    if iteration == max_iterations - 1:
        print('Maximum number of iterations reached without convergence.')
    return P, node_idx
```

7.2 优化稀疏矩阵

7.2.1 数据存储

稀疏矩阵的存储方式与基础 PageRank 的存储方式相同，故在此不再赘述。

7.2.2 PageRank 计算

稀疏矩阵由于其边数远小于顶点数，使用矩阵乘法计算效率会较低，且计算过程中的内存开销较大，因此采用随机游走的方式进行计算。通过随机游走的方式计算有以下两种好处：

- 有效处理大规模图：在网络分析、社交网络、推荐系统等领域，通常会遇到大规模的图数据，这些图数据通常是稀疏的。采用随机游走的方式可以更好地处理这些大规模的图数据，因为它可以

针对图的结构进行局部计算，而不需要考虑整个图的节点和边。

- 自然地考虑随机跳转：随机游走模型自然地考虑了用户或者节点的随机跳转行为。在 PageRank 算法中，随机跳转相当于用户在浏览网页时的随机点击行为，通过引入随机跳转，可以更好地模拟现实世界中的用户行为。

以下为计算步骤：

1. 从图对象中获取所有节点，并计算节点数量。
2. 初始化节点的 PageRank 值为初始值（即 1 - teleport_parameter 除以节点数量）。
3. 然后，进行迭代更新 PageRank 值，直到满足收敛条件（差值小于收敛阈值）。
4. 在每次迭代中，遍历图中的每个节点，根据其邻居节点的 PageRank 值和出链数，更新节点的 PageRank 值。
5. 在更新完所有节点的 PageRank 值后，计算新旧 PageRank 值之差的绝对值之和作为收敛判断依据。
6. 最后，返回计算得到的 PageRank 值字典。

以下为计算代码：

```
def calculate_page_rank(graph, teleport_parameter=0.85,
    convergence_threshold=1e-12):
"""
使用随机游走算法基于给定的图、节点和参数来计算PageRank值。
"""
all_nodes = graph.all_nodes
num_nodes = len(all_nodes)
initial_rank_new = (1 - teleport_parameter) / num_nodes
difference = 1.0
# 初始化节点的 PageRank 值为初始值
old_rank = {node: initial_rank_new for node in all_nodes}
# 迭代更新 PageRank 值，直到收敛
iteration = 1
print("开始迭代")
while difference > convergence_threshold:
    new_rank = {node: initial_rank_new for node in all_nodes}

    # 遍历图中的每个节点，更新 PageRank 值
    for node, neighbors in graph.graph.items():
        # 计算节点的出链数
        num_outgoing = len(neighbors)
        # 更新节点的 PageRank 值
        for neighbor in neighbors:
```

```
                new_rank[neighbor] += teleport_parameter * old_rank[node] /
                    num_outgoing

        s = sum(new_rank.values())
        adjustment = (1 - s) / num_nodes
        new_rank = {k: new_rank[k] + adjustment for k in new_rank}

        # 计算新旧 PageRank 值之差的绝对值之和
        differences = [abs(new_rank[node] - old_rank[node]) for node in all_nodes]
        difference = np.sum(differences)

        old_rank = new_rank

        print('迭代次数:', iteration, '差值:', difference)
        iteration += 1

    print('计算完成.')
    return old_rank
```

7.3 Block Strip

7.3.1 数据预处理

预处理的目标是将数据分割为 <src, degree, <list_of_dest» 的形式。而由于计算 pagerank 时需要分块将 r_new 和稀疏矩阵读入内存，且计算矩阵大小为 block_size，因此，在对数据进行分割时，每一条处理后的数据 <list_of_dest> 所包含的 dest 值必须在一次计算的 block_size 范围内。此外，还可以在预处理中统计结点个数以及每个结点的出度。

具体流程如下所示：

1. 逐行读取数据文件 Data.txt，将 srcNode 和 destNode 存入 edges 列表中
   - edges 列表的形式依旧为 <src, dest>
2. 统计结点个数 N 与出度 out_of_nodes
3. 按 dest 从小到大的顺序给 edges 列表排序
4. edges 列表写入新数据文件 new_data.txt

```
def sort_by_dest(data_path, new_data_path):
    edges = []
    out_of_nodes = defaultdict(int)
    N=set()
    with open(data_path, 'r') as f:
        for line in f:
```

```
            src, dest = map(int, line.strip().split())
            out_of_nodes[src] += 1 #统计出度
            #统计节点个数
            N.add(src)
            N.add(dest)
            if (src, dest) not in edges:
                edges.append((src, dest))

    # 按dest排序
    sorted_edges = sorted(edges, key=lambda x: x[1])
    # 写入新的数据文件
    with open(new_data_path, 'w') as f:
        for edge in sorted_edges:
            f.write(str(edge[0]) + " " + str(edge[1]) + "\n")

    return len(N), out_of_nodes
```

7.3.2 分块

预处理结束后，需要对数据进行分块：根据排序后的 <src, dest> 文件，按照 block_size 的大小分割出 <src, degree, <list_of_dest»。其中，每个结点的 degree 在上一个函数中已经得出。

由于计算矩阵 r^{new} 的大小为 block_size，即每次读取进内存的文件的 dest 范围只能在 block_size 以内。也就是说，第一个文件的 dest 范围为 1 block_size，第二个文件中 dest 范围为 block_size 2 * block_size，以此类推。

具体流程如下所示：

1. 计算需要划分的块数 k
2. 设置字典 temp_save_dict，格式如下所示：
   - key: src
   - value: [degree, [dest_list]]
3. 逐行遍历处理后的数据文件 new_data，获取 src 与 dest 的值
   - 若 dest 在当前块，将 dest 存入 temp_save_dict[src]
   - 若 dest 超出当前块，则将 temp_save_dict 写入新文件后清空，将 dest 存入清空后的字典中
4. 保存最后一个块对应的字典，写入到新文件中

```
def process_data(N, new_data_path, out_of_nodes, block_size) :
    if not os.path.exists(DATA_STRIP):
        os.makedirs(os.path.dirname(DATA_STRIP))
```

```
4
5   # 计算需要划分的块数
6   k = math.ceil(N / block_size)
7
8   temp_save_dict = dict()
9   num = 0
10  with open(new_data_path, 'r') as f:
11      for line in f:
12          src, dest = map(int, line.strip().split())
13
14          #dest超过范围则保存
15          if(dest > (block_size + num * block_size)) :
16              #temp_save_dict = sorted(temp_save_dict.items())
17              temp_save_dict= OrderedDict(sorted(temp_save_dict.items(), key=lambda
                    x: int(x[0])))
18              save_file_name = os.path.join(DATA_STRIP, f'{num}.txt')
19              with open(save_file_name, 'w', encoding='utf-8') as out:
20                  out.write(json.dumps(temp_save_dict))
21              temp_save_dict = dict() #清空
22              num +=1
23
24          #存入临时字典
25          if src not in temp_save_dict:
26              temp_save_dict[src] = [out_of_nodes[src], [dest]]
27          else:
28              temp_save_dict[src][1].append(dest)
29
30  #保存剩余数据
31  if N % block_size != 0:
32      temp_save_dict = OrderedDict(sorted(temp_save_dict.items(), key=lambda x:
            int(x[0])))
33      save_file_name = os.path.join(DATA_STRIP, f'{num}.txt')
34      with open(save_file_name, 'w', encoding='utf-8') as out:
35          out.write(json.dumps(temp_save_dict))
```

7.3.3 PageRank 计算

对于迭代更新 Pagerank 值，需要依次将分割好的数据文件读入内存，并且将对应的 block_size 个 r^{old} 列向量的值读入内存。按照先前所说的公式更新 r^{new} 即可。具体流程如下所示：

1. 遍历每一数据文件，将文件读入内存
2. 将 src 对应的 r^{old} 读入内存

3. 计算 $r^{temp}(dest_j) = b * r^{old}(i)/d_i$
4. 当 dest 的值超过当前块时，将 r^{temp} 写入临时文件中
5. 完成该文件的遍历后，更新 $r^{old} = r^{old} + r^{temp}$
6. 计算 r^{old} 与 r^{new} 的误差 e
7. 将 r^{new} 更新为已重写后的 r^{old}
8. 判断误差 e 的值是否小于特定值 E，是则收敛
    - 若收敛，则终止迭代
    - 若不收敛，则清空 r^{temp} 和 r^{new}，重复上述步骤

Dead Ends 与 Spider Trap 问题

对于 Dead Ends 与 Spider Trap 问题，采用的解决方式为：对每轮更新后的结果 r^{temp} 进行求和，得到 S，并按照如下修正的公式更新到 r^{new} 中：

$r^{new}(dest_j) += b * r^{old}(i)/d_i + (1 - S)/N$

相关代码如下所示:

```
def block_pagerank(size, N):
    # 初始化列向量 r_old
    r_old = {}
    for src in range(1, N+1):
        r_old[src] = np.float64(1/N)

    r_new = np.zeros(N+1)

    if_end = 0
    strip_list = os.listdir(DATA_STRIP) #文件列表
    K = N // size
    if N % size != 0:
        K += 1

    round = 1
    while not if_end:
        S = 0

        # 遍历每一文件 (分割后的块)
        for strip_file in range(len(strip_list)):
            strip_path = DATA_STRIP + str(strip_file) + '.txt' #分割后数据
            with open(strip_path, 'r', encoding='utf-8') as f1:
                strip = json.loads(f1.read()) #读取数据
```

```
            if strip == {}:
                continue
            else: #文件数据不为空
                for src in strip:
                    degree = strip[src][0]
                    r_old_i = r_old[int(src,10)]
                    for dest in strip[src][1]:
                        r_new[dest-0] += b * r_old_i / degree

        e = 0.0

        #处理Dead ends和spider trap
        for i in range(1, N+1):
            S += r_new[i]

        for dest in range(1, N+1):
            r_new[dest] += np.float64((1 - S) / N)


        #最大误差e
        for i in range(1, N+1):
            diff = abs(r_old[i] - r_new[i])
            r_old[i] = r_new[i] #更新r_old
            if e < diff :
                e = diff

        print("count:", round, " e:", e)
        round += 1

        if abs(e) <= E:
            if_end = 1
        else:
            r_new = np.zeros(N+1)

    print("Finish PageRank!\n")
    return r_old
```

8 实验结果

8.1 结果分析

结合本次所写的代码与标准库运行出的结果，我们发现所得到的排名前十的节点次序与标准库所得到的结果基本相吻合。在基础矩阵计算得到的前 100 的结果当中，我们有 22 个不相吻合的排序节点，经过数据分析可发现，不一致顺序的节点之间均属于前后关系，即按照标准库得到的结果为 AB 节点排序，而实验过程中得到 BA 节点的排序。出现错序节点的 PageRank 值较为接近，相差小于 0.00001，中间没有其他节点，证明我们的基础 PageRank 的计算正确率较高。而稀疏矩阵的结果排序与基础块的排序一致，且出现与标准库不一致的地方与基础的相同。Block Strip 分块算法所得到的结果与基础计算的相同。说明了本次三种实现 PageRank 算法的方式具有正确性，结果偏差应当是计算过程中的误差所导致。

图 8.1: 实验结果与标准库结果比较

对于分块算法，从性能角度分析，由于分块算法在切割数据时使用了排序函数，因此切割时会比较慢。而由于 r^{old} 放在内存中进行计算，因此切割完成后的迭代计算的速度较快，总体来说性能不差。在分块的迭代算法中，随机游走因子的计算方式是由 r^{new} 的值累加得到，并非固定值，因此可能会对计算结果造成一定影响。从上述实验结果可知，分块计算结果与其他方式得出的结果大致相同，误差较小，说明该方法可行。

8.2 基础 PageRank 计算结果

Rank	Node ID	Score
1	2730	0.0008718595161773104
2	7102	0.000854534150303149
3	1010	0.0008496162115843206
4	368	0.0008359030857194538
5	1907	0.000830594718331365
6	7453	0.0008206465522624204
7	4583	0.0008178829603525374
8	7420	0.0008103358505538742
9	1847	0.000809998989809893
10	5369	0.000805999584485089

表 1: 基础 PageRank 计算结果

8.3 优化稀疏矩阵 PageRank 计算结果

Rank	Node ID	Score
1	2730	0.000871859516172
2	7102	0.000854534150298
3	1010	0.000849616211579
4	368	0.000835903085715
5	1907	0.000830594718327
6	7453	0.000820646552258
7	4583	0.000817882960348
8	7420	0.000810335850549
9	1847	0.000809998989805
10	5369	0.000805999584480

表 2: 优化稀疏矩阵 PageRank 实验结果

8.4 Block-Strip 运行结果

Rank	Node ID	Score
1	2730	0.000876068651006165
2	7102	0.0008562566382649127
3	1010	0.0008537952524058578
4	368	0.0008391660729973354
5	1907	0.000827954616094344
6	7453	0.0008256732269205815
7	4583	0.0008169193610228027
8	5369	0.0008159851834323679
9	7420	0.0008143939110906898
10	1847	0.0008120186862339357

表 3: 优化稀疏矩阵 PageRank 实验结果

8.5 标准库运行结果

Rank	Node ID	Score
1	2730	0.0008216276997718973
2	7102	0.0008048835476384986
3	1010	0.0008002403594732124
4	368	0.0007874373636582991
5	1907	0.0007822963810319789
6	7453	0.0007741909433914033
7	4583	0.0007708807602648658
8	1847	0.0007640237493953989
9	7420	0.0007633094455394361
10	5369	0.000760127881640054

表 4: 标准库运行结果

9 实验总结

在本次实验中，我们对 PageRank 算法进行了深入的研究和实验。通过实验，我我们深刻理解了 PageRank 算法的原理和工作方式，并掌握了实现该算法的基本步骤和技巧。在实验过程中，我们学会了如何处理图中的 dead ends 和 spider traps 问题等。同时，我们也发现了一些优化算法的方法，例如稀疏矩阵的处理和矩阵乘法的优化，这些方法能够显著提高算法的运行效率和性能。此外，通过分块算法，了解了如何对于大规模数据数据进行处理、存储与运算，提升了对大规模数据集的处理能力，也加深了对于链接分析与 PageRank 算法的认识。

参考文献

[1] Pagerank 算法详解. https://zhuanlan.zhihu.com/p/137561088.