大数据计算及应用实验报告

Lab1 PageRank

2112492	2112338	2213924
刘修铭	李威远	申宗尚

1 题目

对于给定数据集，给出基于 PageRank 分数的前 100 个节点，并尝试不同的参数。必须给出阻尼因子为 0.85 时的结果数据。

除了基本的 PageRank 算法外，还需要完成 Block-Stripe Update 算法。

2 实验要求

- 实现 PageRank 算法，考虑 dead ends 和 spider trap 节点
- 优化稀疏矩阵，提高算法性能
- 实现分块计算，提高计算效率
- 程序需迭代至收敛
- 不可直接调用接口
- 结果格式（.txt 文件）：[NodeID] [Score]

3 实验数据分析

本次实验数据，有向图共有 6110 个 from_node，而所有的节点数为 8297个，由此可见，生成的转移矩阵中存在大量无效数据（0），为后续优化存储提供条件。

共有 6110 个节点出度不为 0，说明存在大量节点只入不出，因此算法实现时需要着重考虑该部分节点。

4 实验原理

4.1 PageRank 算法解释

Pagerank 算法是一种用于评估网络中节点重要性的算法，Pagerank 值反映了一个网页被其他网页链接的数量和质量，即被其他网页认为重要的程度。

Pagerank 算法的核心原理基于以下两个假设：

1. **链接数量假设：** 如果一个网页被很多其他网页链接，那么这个网页可能更重要。这是因为其他网页选择链接到该网页，表明它们认为这个网页的内容或信息是有价值的。

2. **链接质量假设：** 如果一个重要的网页链接到另一个网页，那么被链接的网页可能也很重要。这是因为重要网页的链接可以被视为一种背书，即它们在某种程度上认可被链接网页的内容或信息。

Pagerank 算法通过迭代计算节点的 Pagerank 值来实现。它将网页排名问题转化为一个概率传递问题，其中网页的排名被视为概率分布。算法的基本思想是将网络中的节点视为**随机游走者**，在每一步中，游走者以一定的概率按照链接关系跳转到其他节点。Pagerank 值最终被定义为游走者在长期随机游走后到达每个节点的概率。

算法的基本公式为：

$$PB(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

其中，$PB(v)$ 代表指向 u 节点的其他节点的 PR 值，$L(v)$ 代表指向 u 节点的其他节点的出度数。初始的时候，每个节点被初始化为 $\frac{1}{N}$，N 为节点总数。接着迭代计算，即可求出每个节点的 PageRank 分数。在计算时，通常使用马尔可夫矩阵。使用 M 表示当前 PR 值的矩阵表示，V 表示上一次得到的 PR 值，二者相乘即可得到新的 PR 值。由此不断计算迭代即可。

但是在原始 PageRank 算法中，存在两种特殊节点：dead end 和 spider trap 节点。

- dead end 节点：即，有的节点没有出度，只有入度，会导致 PageRank 变为 0。该问题的解决方案如下：

    1. **随机跳转**：在计算 PageRank 时，引入随机跳转概念。即当一个页面是 dead end 时，随机跳转到其他页面，使得流量能够继续流动，避免了流量损失。这种方法模拟了一个随机的用户行为，增加了网络的连通性。

    2. **人为增加链接**：对于存在 dead end 的页面，可以人为地增加链接，将其链接到其他页面上。这样可以增加页面之间的联系，提高网络的连通性，从而改善 PageRank 的计算结果。

    3. **重新调整转移概率**：在计算 PageRank 时，可以重新调整转移概率，使得即使在存在 dead end 的情况下，页面之间的流量仍然能够流动。可以根据实际情况，对转移概率进行重新分配，以减少 dead end 的影响。

    4. **添加虚拟链接**：对于存在 dead end 的页面，可以添加虚拟链接，将其链接到网络中的其他页面上。这样可以增加页面之间的链接关系，提高网络的连通性，从而改善 PageRank 的计算结果。

- spider trap 节点：部分节点之间形成了一个循环链接，导致在访问这些页面时陷入无限循环，无法终止，最终该部分节点的 PageRank 值趋向为 1，其余逐渐变成 0。该问题的解决方案如下：

    1. **随机跳转**：在计算 PageRank 时，引入随机跳转概念。当遇到 spider trap 时，可以模拟随机用户的行为，随机跳转到其他页面，避免陷入循环链接的无限循环。

    2. **人为干预**：手动调整页面之间的链接结构，断开循环链接，或者在循环链接中添加 nofollow 标签，告知不要跟踪这些链接。

    3. **使用概率调整**：在计算 PageRank 时，对链接转移概率进行调整，以避免陷入循环链接的情况。根据实际情况，对具有循环链接的页面的转移概率进行调整，使得能够正确地跳出循环链接。

基于上述讨论，给出算法的修正公式：

$$PR(u) = \frac{1-\alpha}{N} + \alpha \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

其中，α 被称为阻尼因子，表示一个随机用户按照链接随机浏览页面的概率。通常情况下，α 取值约为0.85，这意味着有 85% 的概率随机用户会继续点击页面上的链接，而 15% 的概率会随机跳转到其他页面。

4.2 PageRank 计算方法

4.2.1 迭代法

迭代法即为不断迭代，直到相邻两次的差值达到设定阈值为止。此处使用向量和矩阵对前面的公式进行进一步推导，并最后得到适用于编程的公式。

对于各个节点的 PageRank 计算矩阵，使用符号进行表示：

$$P_{n+1} = \alpha S P_n + \frac{1-\alpha}{N} e^T$$

提取常数项，得到常数矩阵 A：

$$A = \alpha S + \frac{1-\alpha}{N} e e^T$$

则 PageRank 的计算公式如下：

$$P_{n+1} = A P_n$$

基于此公式即可进行迭代计算。

4.2.2 代数法

基于 PageRank 算法的收敛特性，可以得到如下公式：

$$P = \alpha S P + \frac{1-\alpha}{N} e^T$$

即：

$$P = (E - \alpha S)^{-1} \frac{1-\alpha}{N} e^T$$

由此，求出矩阵的逆即可完成对 PageRank 分数的求解。

4.3 PageRank 算法优化方向

4.3.1 分块计算

在 PageRank 算法中，图的规模可能非常庞大，包含数百万甚至数十亿个节点。对于这样大规模的图，进行一次完整的 PageRank 计算可能会非常耗时，甚至超出单个计算节点的处理能力。分块计算的主要目的是解决大规模图计算的性能瓶颈，提高计算效率。

分块算法是一种将计算任务分割成若干个较小的子任务，分别处理这些子任务并最终合并结果的方法，其基本原理是将整个 PageRank 迭代计算过程分成若干个块，每个块中包含一部分节点的计算，该算法可以充分利用现代计算机的多核处理器和并发计算能力，加速 PageRank 的计算过程。通过将大任务拆分成多个小任务，并行执行，可以显著减少计算时间。同时，由于每个块的计算是独立的，因此易于实现并行化，提高了计算的效率。

4.3.2 优化稀疏矩阵

在 PageRank 算法中，图的规模可能非常庞大，因为大多数节点只与图中的少数几个节点相连，也就导致普通的 PageRank 算法中生成的有向图的邻接矩阵中含有大量 0 元素，即是一个“稀疏矩阵”。稀疏矩阵的优化有如下优点：

1. **减少内存消耗：** 对于大规模的图数据，如果使用密集矩阵表示邻接关系，会浪费大量的内存空间，因为大多数元素都是零。而稀疏矩阵只存储非零元素及其位置信息，可以显著减少内存消耗，提高内存利用率。

2. **降低计算复杂度：** 在稀疏矩阵中，大部分元素都是零，因此在进行计算时可以跳过这些零元素，减少计算量。相比之下，使用密集矩阵进行计算会涉及大量的零元素，增加计算的复杂度。

3. **提高计算效率：** 由于稀疏矩阵的计算量较少，因此可以更快地进行计算。对于 PageRank 算法这样需要进行大量迭代计算的任务来说，稀疏矩阵的优化可以显著提高计算效率，缩短计算时间。

4. **适应大规模数据：** 随着图数据规模的增大，对内存和计算资源的需求也会增加。稀疏矩阵的优化可以使得算法更加适应大规模的图数据，降低资源消耗，提高算法的可扩展性。

4.3.3 TrustRank

4.3.3.1 设计目标

TrustRank 是近年来比较受关注的基于链接关系的排名算法，其可以翻译为“信任指数”。

TrustRank 算法是设计来应对利用垃圾网站轻易操纵 Google 排名、提升搜索结果质量的作弊手段。实施这一方法极大地增加了短时间操作排名的难度，迅速改善了搜索结果的质量。所有要以 TrustRank 值作为网页排名的重要依据，页面的 TrustRank 用来评价其是否具有真正权威性。

4.3.3.2 设计原理

Trustrank 的实现原理基于以下假设：

1. **链接信任传递：** Trustrank 认为，高质量网页通常会链接到其他高质量网页。因此，当一个网页被其他高质量网页链接时，它也会因此获得一定的信任值。

2. **反垃圾链接：** Trustrank 也考虑到垃圾网页链接的影响。如果一个网页主动链接到大量的垃圾网页或被垃圾网页链接，则其信任值会相应下降。

基于以上原理，Trustrank 算法可以概括为以下几个步骤：

1. **初始化：** 人工选取具有百分百信任度的网页作为**种子向量**。

2. **链接分析：** 算法分析所有网页之间的链接关系。它检查每个网页的出链和入链，并考虑这些链接的来源和目标网页的信任值。

3. **信任传递：** 通过链接关系，信任值从高质量网页传递到链接的网页。如果一个网页被高信任值网页链接，则其信任值会相应提高。

4. **反垃圾链接处理：** 算法考虑到反垃圾链接的影响，即网页链接到垃圾网页的情况。这种情况会导致网页的信任值下降。

5. **迭代计算：** 以上过程迭代进行，直到收敛或达到预定的迭代次数。

6. **排名计算：** 最终，根据网页的信任值进行排名。信任值高的网页会在搜索结果中获得更高的排名。

因此，相较于 Pagerank，Trustrank 的更新方式如下：

$$r = \alpha \times T \times r + (1 - \alpha) \times d$$

其中 d 为种子向量，r 为每次的可信度向量，初始值为种子向量。

4.3.3.3 结果评价

在我们的实验中，如何实现 TrustRank 的性能指标评价呢？我们结合推荐系统中常见的评价指标，设计了指标 IoR：

$$IoR_M = \frac{1}{|U|} \sum_{u=1}^{|U|} -(R(n_u|s_{TR}) - R(n_u|s_{PR}))$$

该指标中，集合 U 为垃圾网站集合，即所有垃圾网站在 Trustrank 排序下和 Pagerank 排序下排名的平均变化值，基于此，我们可以较好地看出 Trustrank 能否完成对垃圾网站的可信度抑制。

5 实验过程（含关键代码解析）

5.1 原始 PangeRank 算法

1. 首先定义了 `read_graph` 函数，将给定的数据文件读取为有向图的形式，并提取出所有的节点到集合 `nodes` 中。

```python
def read_graph(file_path):
    G = {}
    nodes = []
    with open(file_path, 'r') as f:
        for line in f:
            from_node, to_node = map(int, line.strip().split())
            if from_node not in G:
                G[from_node] = []
            G[from_node].append(to_node)
            if from_node not in nodes:
                nodes.append(from_node)
            if to_node not in nodes:
                nodes.append(to_node)
    return G, nodes
```

2. 接着定义了 `output_result` 函数，用于将提取出的数据按照格式要求写入到输出文件中。

```python
def output_result(results, file_path):
    with open(file_path, 'w') as f:
        for node, score in results:
            f.write(f"{node} {score}\n")
```

3. 下面对 PageRank 核心计算部分进行解释。

- 首先对于前面提取出的节点集合进行处理，按照节点的序号进行排序标号，得到其索引集合，便于后续处理。

```python
index = {}
for i, node in enumerate(sorted(nodes)):
    index[node] = i
```

- 接着按照前面构建的有向图，结合构建的索引，构建转移矩阵 S。

```
S = np.zeros([N, N], dtype = np.float64)
for from_node, to_nodes in G.items():
    for to_node in to_nodes:
        S[index[to_node], index[from_node]] = 1
```

- 然后对 S 进行初始化。此处实现时考虑了 dead end 和 spider trap 问题。

```
for j in range(N):
    sum_of_col = sum(S[:, j])
    if sum_of_col == 0:
        S[:, j] = 1 / N
    else:
        S[:, j] /= sum_of_col
```

- 按照前面**实验原理**中解释的方法对其进行计算。下面将按照不同的方法对其分别进行解释。

- 最后基于前面得到的 PageRank 值，排序后，使用索引输出最后结果。

```
sorted_nodes = sorted(index.items(), key = lambda x: P_n[x[1]], reverse=True)

sorted_results = []
for node, index in sorted_nodes[:100]:
    sorted_results.append((node, P_n[index]))
```

5.1.1 迭代法实现

./codes/basic_method1.py

首先构建常数矩阵 A。

```
A = teleport_parameter * S + (1 - teleport_parameter) / N * np.ones([N, N], dtype = np.float64)
```

接着初始化两个矩阵、误差以及阈值用于后续迭代计算。

```
P_n = np.ones(N, dtype = np.float64) / N
P_n1 = np.zeros(N, dtype = np.float64)
e = 100
tol = 1 / (N * N)
```

最后使用循环进行迭代计算，当误差在阈值允许的范围内即停止。

```
while e > tol:
    P_n1 = np.dot(A, P_n)
    e = P_n1 - P_n
    e = max(map(abs, e))
    P_n = P_n1
```

5.1.2 代数法实现

./codes/basic_method2.py

首先创建两个矩阵，一个为大小为 $N \times N$ 的单位阵，另一个为大小为 $N \times 1$ 的列向量。

```
e = np.identity(N, dtype = np.float64)
eT = np.ones([N, 1], dtype = np.float64)
```

接着按照前面推导的公式，求解出 PageRank 值。

```
P = np.dot(np.linalg.inv(e - teleport_parameter * S), ((1 - teleport_parameter) / N *
eT)).flatten()
```

5.2 PageRank 算法优化

5.2.1 分块计算

5.2.1.1 多线程尝试

5.2.1.1.1 threading

./codes/extension_block_threading.py

基础实现中已将数据处理函数进行解释，此处仅对核心代码进行说明

为了进一步优化程序的运行时间，注意到分块之后的矩阵 M 含有大量密集计算任务，因此想进行程序多线程运行方向的尝试，从而当程序需要进行 I/O 操作（主要是文件读写），多线程运行可以使得程序在等待 I/O 操作完成时能够继续执行其他任务，提高了程序的效率。

- Python 支持多线程编程，只需要导入自带的 threading 模块

```
import threading
```

由于在 python 的 threading 库中，thread 对象并不存在返回值，所以小组自己设计了一个 MyThread 类，通过 get_result 成员函数获取函数的返回值，从而使用 threading 进行函数的运行。

```
class MyThread(threading.Thread):
    def __init__(self, func, args=()):
        super(MyThread, self).__init__()
        self.func = func
        self.args = args
    def run(self):
        time.sleep(2)
        self.result = self.func(*self.args)
    def get_result(self):
        threading.Thread.join(self)  # 等待线程执行完毕
        try:
            return self.result
        except Exception:
            return None
```

核心算法如下：

```
while e > tol:
        print(e)
        mythreads = []
        for block_start in range(0, N, block_size):
            block_end = min(block_start + block_size, N)
            mythreads.append(
                MyThread(iter_once,(G,P_n,block_start,block_end,teleport_parameter,N))
            )
        for mythread in mythreads:
            mythread.start()
        P_n1 = np.concatenate([mythread.get_result() for mythread in mythreads])
        e = P_n1 - P_n
        e = max(map(abs, e))
        P_n = P_n1
```

通过构建 mythreads 列表，将每一步迭代需要执行的函数名 `iter once` 和参数 `G,P_n,block_start,block_end,teleport_parameter` 构建出新的 MyThread 对象，然后在 mythreads 列表中将所有的 mythread 对象进行多线程运行，通过 `get_result` 获取返回值，使用 `np.concatenate()` 函数将其拼为新的 P_n1，进行新一轮迭代。

5.2.1.1.2 ThreadPoolExecutor

./codes/extension_block_ ThreadPoolExecutor.py

基础实现中已将数据处理函数进行解释，此处仅对核心代码进行说明

除了使用 threading 外，小组还尝试了使用 ThreadPoolExecutor 进行多线程并行计算。使用了 `concurrent.futures` 模块的 `ThreadPoolExecutor` 来创建线程池，并且使用了 `executor.submit()` 方法来提交任务，实现了并行计算的效果。具体来说，每个块的计算任务被提交到线程池中，并且在有空闲线程时会立即执行，因此这部分代码是并行执行的。其核心代码如下：

```
while e > tol:
    print(e)
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = []
        for block_start in range(0, N, block_size):
            block_end = min(block_start + block_size, N)
            futures.append(executor.submit(iter_once, G, P_n, block_start, block_end,
teleport_parameter, N))
        P_n1 = np.concatenate([future.result() for future in futures])
    e = P_n1 - P_n
    e = max(map(abs, e))
    P_n = P_n1
```

然而，Python 含有一个全局解释器锁 GIL。

由于 Python 代码的执行是由 Python 虚拟机（又叫解释器主循环）进行控制的，因此其设计思路是在主循环中同时只能有一个控制线程在执行，就像单核 CPU 系统中的多进程一样，只是在宏观上看起来多进程，在微观上其实仍是单个单个的运行。同理，尽管 Python 解释器中可以有多个线程在运行，但是微观时间上，只有一个线程在被真正执行。GIL 便是控制 Python 虚拟机访问的部件，保证**同时只能有一个线程在进行**。

因此，对于 I/O 密集型的任务，使用 `threading` 模块可以有效缩短程序运行时间，然而对于 CPU 密集型的任务(需要大量的 CPU 计算资源)，使用多线程运行，不仅不会优化程序，反而会加长代码的执行时间。值得指出的是，在本题的每次循环中，进行了大量的数值运算，比起 I/O 操作，CPU 的使用率远远更高，因此，使用多线程是一次错误的尝试。

5.2.1.2 多进程尝试

./codes/extension_block_multiprocess.py

基础实现中已将数据处理函数进行解释，此处仅对核心代码进行说明

在多线程的尝试失败后，通过学习发现了多进程的程序执行方式。即每个进程都有自己独立的内存空间和 Python 解释器，它们之间不共享内存，从而并不会产生多线程下多个线程在同一时刻竞争 GIL 的情况，因此更加适用于 CPU 密集型任务。

进程池是资源进程, 管理进程组成的技术的应用，通过使用进程池，可以更加高效地完成多进程程序的执行。

使用多进程程序，需要导入 `multiprocess` 库，同时，使用进程池，导入 `concurrent.futures` 库：

```
from concurrent.futures import ProcessPoolExecutor,as_completed
from multiprocessing import Process
```

核心算法更改如下：

```
 while e > tol:
        print(e)
        with ProcessPoolExecutor() as executor:
            futures = []
            for block_start in range(0, N, block_size):
                block_end = min(block_start + block_size, N)
                futures.append(executor.submit(iter_once, G, P_n, block_start, block_end,
teleport_parameter, N))
            P_n1 = np.concatenate([future.result() for future in futures])
        e = P_n1 - P_n
        e = max(map(abs, e))
        P_n = P_n1
```

首先定义 `concurrent.futures` 中的 `ProcessPoolExecutor` 作为 `executor`，然后创造空 `futures` 列表，将需要执行的函数 `iter_once` 和参数通过 `executor.submit()` 传入到 futures 列表中，随后直接使用 future 的 `result()` 函数，通过和前面一样的方式，得到最终多线程运算的结果。

5.2.2 优化稀疏矩阵

./codes/extension_sparse.py

基础实现中已将数据处理函数进行解释，此处仅对核心代码进行说明

在此种实现中，不创建转移矩阵，而是利用图的 hash 表存储转移矩阵进行计算。即虽然代码中并没有显式地构建整个转移矩阵，但是在计算过程中利用了图的邻接表结构隐式地表示了转移矩阵中的非零元素。这种隐式构建方式避免了存储大量的零元素，节省了内存空间。

在函数 `iter_once` 中，对于每个节点的 PageRank 值计算，都利用了稀疏性质。在计算每个节点的 PageRank 时，并不需要遍历整个图，而是根据节点的邻接表信息进行计算。如果节点 i 有邻居节点，那么它的 PageRank 值会受到邻居节点的贡献，否则它的 PageRank 值仅由随机跳转参数和总节点数决定。函数代码如下：

```
def iter_once(G, P_n, N, teleport_parameter):
    P_n1 = np.zeros(N, dtype=np.float64)
    for i in range(N):
        if i in G.keys():
            exist = np.isin(np.arange(N), G[i])
            P_n1 += ((teleport_parameter * exist / len(G[i]) + (1 - teleport_parameter) / N)) *
P_n[i]
        else:
            P_n1 += P_n[i] / N
    return P_n1
```

5.2.3 TrustRank

./codes/TrustRank/goodWebSearch.py

./codes/TrustRank/TrustRankTest.py

./codes/TrustRank/checkResult.py

基础实现中已将数据处理函数进行解释，此处仅对核心代码进行说明

在尝试对 TrustRank 进行实现的过程，我们首先需要明确的是，如何模拟一个专家，去人工选择起初的种子向量呢？

回顾 TrustRank 的最基本假设：**好的网站很少会链接到坏的网站。**

基于此原理，我们在有限的数据集下，无法判定可信的网站。但是若**假设**某些网站为可信网站，我们可以通过对这些网站周边进行搜索，未被可信网站链接到、且入度较低的网站，具有很大可能为垃圾网站。

因此，我们可以借助这个特性来验证 Trustrank 算法的性能，实验中，我们采取如下的数据处理步骤：

1. **随机选取可信网站：**随机选取 100 个入度较大的网站节点，假定他们为可信网站，并输出到 good.txt 中。
2. **计算垃圾网站：**在可信网站周边进行搜索，选取未被可信网站链接到、且入度较低的网站作为垃圾网站，并输出到 bad.txt 中。
3. **计算排名结果：**使用可信网站生成种子向量，利用 Trustrank 算法进行迭代，输出在 Trustrank 影响下的排名。
4. **指标评价：**将 Trustrank 影响下的排名与 Pagerank 下的排名进行对比，输出计算出的 IoR 指标，观察 Trustrank 效果。

接下来，我们分如上四个方面来介绍 Trustrank 的代码实现：

5.2.3.1 随机选取可信网站

我们对 read_graph 函数进行修改，使其还记录了每个节点的出入度信息：

```python
def read_graph(file_path):
    # 读取图数据并构建图结构
    G = {}
    nodes = []
    in_degrees = {}
    with open(file_path, 'r') as f:
        for line in f:
            ……
            if to_node not in in_degrees:
                in_degrees[to_node] = 0
            in_degrees[to_node] += 1
    return G, nodes, in_degrees
```

基于此，我们可以通过筛选入度为前 5% 的所有节点，并随机选择 100 个将其输出

```python
    # 找到入度最高的前5%的节点
    sorted_nodes = sorted(in_degrees.items(), key=lambda x: x[1], reverse=True)
    top_5_percent = int(len(sorted_nodes) * 0.05)
    top_nodes = [node for node, _ in sorted_nodes[:top_5_percent]]

    # 随机选择100个节点
    random.shuffle(top_nodes)
    selected_nodes = top_nodes[:100]

    # 将选定的节点存储在数组中
    selected_nodes_array = []
    for node in selected_nodes:
        selected_nodes_array.append(node)

    # 将选定的节点输出到test.txt文件
    output_result(selected_nodes_array, output_file_path, in_degrees)
```

5.2.3.2 计算垃圾网站

随后，我们在图中进行检索，寻找与可信网站节点不相邻且入度较小的节点，作为考察的垃圾网站，这里仍然随机选取了一部分：

```python
def find_low_in_degree_nodes(G, nodes, in_degrees, target_nodes, num_nodes):
    # 找到与目标节点不相邻且入度较小的节点
    low_in_degree_nodes = []
    for node in nodes:
        if node not in target_nodes:
            is_adjacent = False
            for target_node in target_nodes:
                if target_node in G and node in G[target_node]:
                    is_adjacent = True
                    break
            if not is_adjacent and in_degrees.get(node, 0) < 9:
                low_in_degree_nodes.append(node)
    random.shuffle(low_in_degree_nodes)
    selected_nodes = low_in_degree_nodes[:num_nodes]
    return selected_nodes
```

5.2.3.3 Trustrank 处理

Trustrank 和 Pagerank 唯一的区别，在于其迭代过程依赖于种子向量（随机跳转只会转移到可信网站上），且其初始值为种子向量。因此需要修改整个的迭代过程如下：

基于如下处理，我们就完成了 Trustrank 下的转移更新步骤，可以看到与 Trustrank 的更新公式是一致的。

```
    # 初始化TrustRank向量T_n(种子向量)
    T_n = np.zeros(N, dtype=np.float64)
    for node in seed_vector:
        T_n[index[node]] = 1 / len(seed_vector)

    D = T_n

    # 构建TrustRank迭代矩阵A
    A = teleport_parameter * S

    # 设置迭代停止条件
    e = 100
    tol = 1 / (N * N)

    # 迭代计算TrustRank
    while e > tol:
        T_n1 = np.dot(A, T_n)  + (1 - teleport_parameter) * D
        e = T_n1 - T_n
        e = max(map(abs, e))
        T_n = T_n1
```

5.2.3.4 指标评价

将 Trustrank 输出的排名与 Pagerank 输出的排名进行对比，并结合之前确定的垃圾网站，计算排名变化指标 IoR：

这里，由于实验选取的节点具有随机性，我们进行多次如上三个步骤，每次都进行评分，取平均值作为 Trustrank 的性能指标。

```
iters = 10

for i in range(iters):
    TrustRankTest.TR()

    # 从bad.txt文件中读取节点
    with open('bad.txt', 'r') as file:
        nodes = file.read().splitlines()

    # 计算每个节点的排名变化
    ranking_changes = []
    for node in nodes:
        with open('./compare/networkx.txt', 'r') as networkx_file,
open('./compare/trustrank.txt', 'r') as trustrank_file:
            networkx_ranking = [line.split()[0] for line in networkx_file]
            trustrank_ranking = [line.split()[0] for line in trustrank_file]

        networkx_index = networkx_ranking.index(node)
```

```
        trustrank_index = trustrank_ranking.index(node)

        ranking_change = trustrank_index - networkx_index
        ranking_changes.append(ranking_change)

    # 计算平均排名变化
    average_change = sum(ranking_changes) / len(ranking_changes)

    # 打印平均排名变化
    print("Average ranking change:", average_change)

    # 将排名变化写入result.txt文件
    with open('result.txt', 'w') as result_file:
        for change in ranking_changes:
            result_file.write(str(change) + '\n')
```

直接执行 checkResult.py 文件，即可进行多次 TrustRank 的评估：

5.3 第三方库实现

为了对自己编写的算法进行评估分析，小组使用第三方库 NetworkX 完成了 PageRank 算法的实现。

- `read_graph` 函数：该函数用于从给定的文件中读取图数据，其中读取部分基于给定的文件格式实现。使用 NetworkX 提供的 `DiGraph` 类创建一个有向图对象。遍历 `Data.txt` 文件，解析出每条边的起始节点和结束节点，并将其添加到有向图中。

```
import networkx as nx
def read_graph(file_path):
    G = nx.DiGraph()
    with open(file_path, 'r') as f:
        for line in f:
            from_node, to_node = map(int, line.strip().split())
            G.add_edge(from_node, to_node)
    return G
```

- `calculate_pagerank_and_sort` 函数：计算图中每个节点的 Pagerank 值，并按值从大到小排序。调用 NetworkX 提供的 `pagerank` 函数来计算 Pagerank 值。NetworkX 库提供的 `pagerank` 函数可以设置阻尼因子、收敛阈值等参数。本次实验中，小组设置阈值为 $\frac{1}{N \times N}$ 分别运行，得到数据用于分析对比。计算完成后，筛选得分前 100 名的节点输出到文件中。

```
def calculate_pagerank_and_sort(G, teleport_parameter=0.85, tol=1e-20, top_n=100):
    pagerank_scores = nx.pagerank(G, alpha=teleport_parameter, tol=tol)
    sorted_scores = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_scores[:top_n]
```

- `output_result` 函数：将排序后的 Pagerank 结果输出到文件。

```
def output_result(results, file_path):
    with open(file_path, 'w') as f:
        for node, score in results:
            f.write(f"{node} {score}\n")
```

5.4 数据对比

./codes/compare.py

为了方便数据对比，小组编写了一个简单的算术程序，将两个输出文件逐行逐项进行做差，并将结果输出到新的 txt 文件中。程序代码如下：

```python
import os

def compare_files(file1_path, file2_path):
    file1_name, file1_ext = os.path.splitext(os.path.basename(file1_path))
    file2_name, file2_ext = os.path.splitext(os.path.basename(file2_path))

    output_file_name = f"{file1_name}_{file2_name}_comparison{file1_ext}"
    output_file_path = os.path.join(os.path.dirname(file1_path), output_file_name)

    with open(file1_path, 'r') as f1, open(file2_path, 'r') as f2, open(output_file_path, 'w') as output_file:
        for line1, line2 in zip(f1, f2):
            num1_1, num1_2 = map(float, line1.strip().split())
            num2_1, num2_2 = map(float, line2.strip().split())
            result1 = num1_1 - num2_1
            result2 = num1_2 - num2_2

            output_file.write(f"{result1} {result2}\n")

if __name__ == "__main__":
    file1_path = input("Enter path for the first file: ")
    file2_path = input("Enter path for the second file: ")

    compare_files(file1_path, file2_path)
```

6 实验结果及分析

6.1 节点排名数据

6.1.1 基于第三方库

基于第三方库 NetworkX 实现的 PageRank 算法得到的结果位于 `./codes/output/networkx.txt` 中。此处以前十名为例进行展示。

```
 1   2730 0.0008711604926536878
 2   7102 0.0008538432149217737
 3   1010 0.0008489290996183523
 4   368 0.0008352286390791629
 5   1907 0.0008299226010093124
 6   7453 0.0008200000782599075
 7   4583 0.0008172288800131311
 8   7420 0.0008096814333798858
 9   1847 0.0008093592005915353
10   5369 0.0008053612360905464
```

6.1.2 基础算法实现

使用自己编写的**基础算法**进行计算，分别使用**迭代法**与**代数法**进行计算，得到两个输出文件 `./codes/output/basic_1.txt` 和 `./codes/output/basic_2.txt`，将二者的数据进行对比，得到结果文件 `./codes/output/basic_1_basic_2_comparison.txt`，可以看到，节点的排名完全相同，且分数误差数量级较小，在合理范围内。

codes > output > basic_1_basic_2_comparison.txt

```
 1   0.0 -4.6986557540487466e-08
 2   0.0 -4.64428935002684e-08
 3   0.0 -4.6185893372817366e-08
 4   0.0 -4.5334562869003316e-08
 5   0.0 -4.5177992110258074e-08
 6   0.0 -4.3454314334876766e-08
 7   0.0 -4.39655926744208e-08
 8   0.0 -4.3988233822708604e-08
 9   0.0 -4.300498039058395e-08
10   0.0 -4.290813193739019e-08
11   0.0 -4.333471915367922e-08
12   0.0 -4.358671682619126e-08
13   0.0 -4.2780824546090335e-08
14   0.0 -4.2598704559681025e-08
15   0.0 -4.1749521378876484e-08
16   0.0 -4.162908125691662e-08
17   0.0 -4.154735973695752e-08
18   0.0 -4.1162557745796566e-08
19   0.0 -4.071469736149306e-08
20   0.0 -4.051314118154372e-08
21   0.0 -4.082543688245337e-08
22   0.0 -3.997757178238453e-08
```

将自己实现的 PageRank 算法与添加阈值的使用第三方库实现的进行对比，得到结果文件 `./codes/output/basic_1_networkx_comparison.txt`，可以看到，排名完全相同，但分数相差数量级较小，在合理范围内，说明小组编程实现的正确性。因此，后面的数据对比中使用该数据作为对比版即可。此处仅展示前十名的对比结果，完整数据可移步输出文件查看。

lab1 > codes > output > basic_1_networkx_comparison.txt

```
  0.0 6.520369662981525e-07
  0.0 6.44492488101789e-07
  0.0 6.409260728103359e-07
  0.0 6.291120776343202e-07
  0.0 6.269393301636637e-07
  0.0 6.03019688388026e-07
  0.0 6.101147469432392e-07
  0.0 6.104289403772046e-07
  0.0 5.96784238167088e-07
  0.0 5.954402628140969e-07
```

6.1.3 稀疏矩阵优化算法

使用进行**稀疏矩阵**优化的算法进行计算，得到其输出结果 `./codes/output/sparse.txt`，将输出结果与前面得到的数据进行对比，得到结果文件 `./codes/output/basic_1_sparse_comparison.txt`，可以看到，排名完全相同，但分数相差数量级较小，在合理范围内，说明小组编程实现的正确性。此处仅展示前十名的对比结果，完整数据可移步输出文件查看。

lab1 > codes > output > basic_1_sparse_comparison.txt

```
  0.0 -4.662069341687669e-17
  0.0 -5.5077470362263625e-17
  0.0 -4.4994390158148434e-17
  0.0 -4.96564594998361e-17
  0.0 -4.716279450311944e-17
  0.0 -5.074066167232161e-17
  0.0 -4.607859233063394e-17
  0.0 -5.0198560586078855e-17
  0.0 -4.586175189613684e-17
  0.0 -4.640385298237959e-17
```

6.1.4 分块计算

6.1.4.1 ThreadPoolExecutor

运行对应代码，得到其对应的输出 `./codes/output/block_ThreadPoolExecutor.txt`，将其与前面得到的数据进行对比，得到结果文件 `./codes/output/basic_1_block_ThreadPoolExecutor_comparison.txt`，可以看到，排名完全相同，但分数相差数量级较小，在合理范围内，说明小组编程实现的正确性。此处仅展示前十名的对比结果，完整数据可移步输出文件查看。

lab1 > codes > output > basic_1_block_ThreadPoolExecutor_comparison.txt

```
  0.0 -4.662069341687669e-17
  0.0 -5.5077470362263625e-17
  0.0 -4.4994390158148434e-17
  0.0 -4.96564594998361e-17
  0.0 -4.716279450311944e-17
  0.0 -5.074066167232161e-17
  0.0 -4.607859233063394e-17
  0.0 -5.0198560586078855e-17
  0.0 -4.586175189613684e-17
  0.0 -4.640385298237959e-17
```

6.1.4.2 threading

运行对应代码，得到其对应的输出 `./codes/output/block_threading.txt`，将其与前面得到的数据进行对比，得到结果文件 `./codes/output/basic_1_block_threading_comparison.txt`，可以看到，排名完全相同，但分数相差数量级较小，在合理范围内，说明小组编程实现的正确性。此处仅展示前十名的对比结果，完整数据可移步输出文件查看。

lab1 > codes > output > basic_1_block_threading_comparison.txt

```
1    0.0 -4.662069341687669e-17
2    0.0 -5.5077470362263625e-17
3    0.0 -4.4994390158148434e-17
4    0.0 -4.96564594998361e-17
5    0.0 -4.716279450311944e-17
6    0.0 -5.074066167232161e-17
7    0.0 -4.607859233063394e-17
8    0.0 -5.0198560586078855e-17
9    0.0 -4.586175189613684e-17
10   0.0 -4.640385298237959e-17
```

6.1.4.3 multiprocess

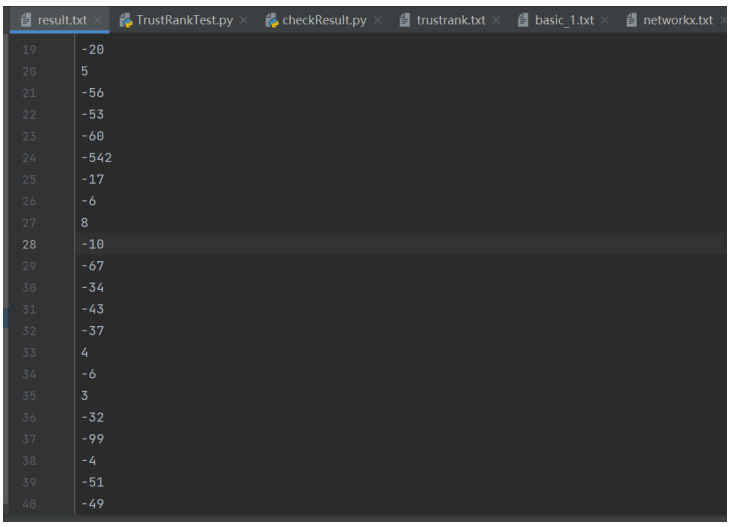
运行对应代码，得到其对应的输出 `./codes/output/block_multiprocess.txt`，将其与前面得到的数据进行对比，得到结果文件 `./codes/output/basic_1_block_multiprocess_comparison.txt`，可以看到，排名完全相同，但分数相差数量级较小，在合理范围内，说明小组编程实现的正确性。此处仅展示前十名的对比结果，完整数据可移步输出文件查看。

lab1 > codes > output > basic_1_block_multiprocess_comparison.txt

```
1    0.0 -4.662069341687669e-17
2    0.0 -5.5077470362263625e-17
3    0.0 -4.4994390158148434e-17
4    0.0 -4.96564594998361e-17
5    0.0 -4.716279450311944e-17
6    0.0 -5.074066167232161e-17
7    0.0 -4.607859233063394e-17
8    0.0 -5.0198560586078855e-17
9    0.0 -4.586175189613684e-17
10   0.0 -4.640385298237959e-17
```

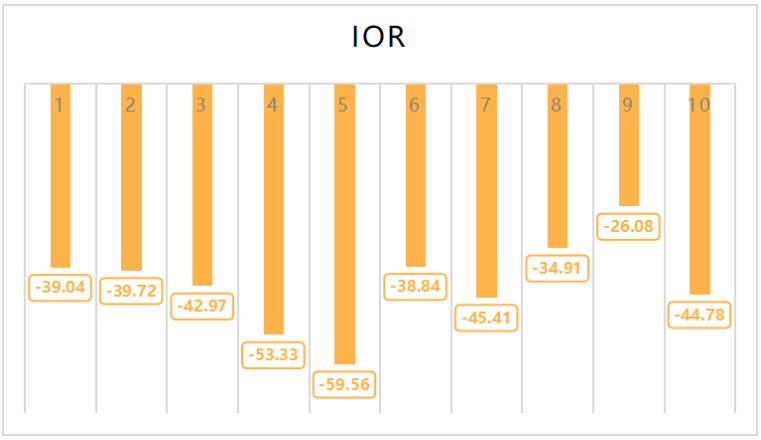
6.1.5 TrustRank

运行评价代码，得到垃圾网站的排名变化如下所示：



可以看到，垃圾网站的经过 Trustrank 处理后，基本上都下降了许多，我们结合 IoR 指标结果进行查看（省略到整数）：

iters	1	2	3	4	5	6	7	8	9	10
IoR	-39	-40	-43	-53	-59	-39	-45	-26	-35	-44



即，Trustrank 执行后，垃圾网站排名变化指标 (IoR) 平均为 -42.464，且均为负数，表明在 Trustrank 作用下，垃圾网站几乎均处于下降趋势，平均每一个垃圾网站都下降了42名，**即说明 Trustrank 算法对垃圾网站有较好的抑制作用**。

6.2 算法运行性能说明

在 `Apple M2 Pro 16G` 环境下运行各个代码文件，得到如下的运行性能数据。

Type	Networkx	Basic_1	Basic_2	ThreadPool	Threading	Multiprocess	Sparse	TrustRank
Storage (MB)	111.1875	1079.2031	36.4219	41.3125	22.0000	67.5000	47.2969	1099.4180
Time (s)	2.7869	4.2044	12.5273	1516.9718	2031.6597	140.6871	11.9909	5.3397

可以看出如下重要信息：

- Pagerank 基本方法具有较高的空间复杂度，在使用稀疏矩阵优化后从 1080MB 的空间占用变为了 47.3MB 的空间占用，表明**稀疏矩阵有较好的空间优化作用。**

- 使用矩阵分块计算虽然能进一步降低空间复杂度，但是会造成时间复杂度的巨额增长，借助多线程并行**优化了这一点不足，其中 Multiprocess 下优化效果最佳。**

7 实验总结与感悟

本次实验中，小组编程实现了 PageRank 算法，并对其进行了如下多个方面的优化：

- **完整充分实现了稀疏矩阵存储优化**，成功大大减小了程序空间复杂度。
- **完整充分实现了矩阵分块更新算法**，进一步减小空间复杂度，并采用 ThreadPoolExecutor、threading、multiprocess 多种多线程并行计算方式进行了时间复杂度的提升与优化。
- **额外实现了迭代法的 Pagerank 算法**，这种方法具有较小的空间复杂度，是另一种 Pagerank 的实现思路。
- **额外实现了 TrustRank 算法**，探索了 Trustrank 对垃圾网站的抑制作用，并提出一种基于排名的评价指标，在数据集上取得了较好的表现。

经过与第三方库得到的结果进行对比，以及相关评价指标的设计评估，可以看到小组实现的效果较好，PageRank 算法的实现度较高。

最后，感谢杨老师的耐心讲授，小组成员均有了较大收获。

8 运行说明

本次实验代码均位于 `./codes` 文件夹中，其中可执行文件 exe 均位于 `./codes/dist` 文件夹中，建议使用 `PowerShell` 运行，运行完成后会输出运行的内存使用情况和时间开销，并在 `./codes/dist/output` 中输出结果文件。

经测试，所有的 exe 文件均可正常运行。但需要注意的是，由于 `./codes/dist/extension_block_multiprocess.exe` 是多进程运行（打包时限定进程数为 10），其对于运行环境的要求较高，容易爆内存导致运行失败，建议直接运行 `./codes/extension_block_multiprocess.py` 文件，此种方法不会出现运行失败的情况（好神奇）。

9 参考

https://blog.csdn.net/apollo_miracle/article/details/117563650

https://blog.csdn.net/qq_41427834/article/details/110262036

https://zhuanlan.zhihu.com/p/189848778

TrustRank算法 - 知乎 (zhihu.com)

PageRank的变体算法：TrustRank、ItemRank和TextRank（三） - 知乎 (zhihu.com)

Hadoop-MapReduce下的PageRank 矩阵分块算法_pagerank 分块计算-CSDN博客

基于六度分隔理论、PageRank等的人工风控特征提取框架-SocialWatch - 知乎 (zhihu.com)