PageRank 实验报告

苏畅 124033910082

2025 年 5 月 21 日

一 PageRank 方法介绍

PageRank 是 Google 提出的网页排序算法，其基本思想是模拟 “随机浏览者” 在网页之间跳转，基于网页之间的链接结构来衡量每个网页的重要性。数学表达如下：

$$PR(i) = \frac{1-\alpha}{N} + \alpha \sum_{j \in M(i)} \frac{PR(j)}{L(j)}$$

其中：

- $PR(i)$ 表示节点 i 的 PageRank 值；
- α 是阻尼系数，默认取 0.85；
- N 是总节点数；
- $M(i)$ 是指向节点 i 的节点集合；
- $L(j)$ 是节点 j 的出度；

对于悬空节点（dangling node，即没有出边的节点），PageRank 的处理方式是将其看作可以随机跳转到任意其他节点，以保证迭代过程中的连通性和收敛性。

二 朴素 PageRank 方法的实现

我首先实现了一个基于原生 Python 数据结构的朴素 PageRank 方法，其核心思想是模拟网页之间通过超链接形成的随机浏览过程，利用迭代求解每个网页的 PageRank 值。具体实现流程如下：

- 从 CSV 文件中读取网页之间的有向边关系，构建有向图结构；
- 使用字典 `dict` 构建邻接表表示图的出边信息，并统计每个节点的出度；
- 所有节点的初始 PageRank 值设为 $1/N$，其中 N 为节点总数；
- 在每轮迭代中，依次更新每个节点的 PageRank 值，更新公式如下：

$$PR^{(t)}(i) = \frac{1-\alpha}{N} + \alpha \sum_{j \in \text{In}(i)} \frac{PR^{(t-1)}(j)}{\deg_{out}(j)} + \frac{\alpha}{N} \cdot \sum_{j \in \text{Dangling}} PR^{(t-1)}(j)$$

其中 α 为阻尼系数（本实验中设为 0.85），$\mathrm{In}(i)$ 表示指向节点 i 的所有节点集合，Dangling 表示所有无出边节点的集合；

- 为了避免陷入循环或被 dangling node 卡死的情况，在每轮中将 dangling 节点的贡献平均分配给所有节点；
- 通过计算两次迭代之间的 L1 范数（即 $\sum_i |PR_i^{(t)} - PR_i^{(t-1)}|$）判断是否收敛，当该值小于 10^{-6} 时认为收敛；
- 收敛后输出所有节点的 PageRank 值，按从高到低排序，最终输出排名前 1000 的节点。

2.1 转移矩阵的构建与存储方式

在朴素实现中，并未直接构建 $N \times N$ 的转移概率矩阵 P，而是使用 Python 的 `dict` 和 `list` 来构建邻接表并维护每个节点的出链信息。每轮迭代中通过这些信息动态地计算出每个节点的 PageRank 值，从而避免了对稠密矩阵的存储开销。

具体而言，对于每一个节点 u，维护其出链节点列表 `out_links[u]` 和出度 `out_degree[u]`，从而在每一轮迭代中将 PR 值均匀分发到出链节点上：

```
for u in range(N):
    if out_degree[u] == 0:
        continue
    share = pr[u] / out_degree[u]
    for v in out_links[u]:
        new_pr[v] += damping * share
```

Listing 1: 朴素方法中转移概率计算方式

此外，对于没有出边的 悬挂节点（*Dangling Nodes*），将其 PageRank 值在下一轮中平均分配给所有节点，以保证概率质量守恒：

```
dangling_sum = sum(pr[u] for u in range(N) if out_degree[u] == 0)
for i in range(N):
    new_pr[i] += damping * dangling_sum / N
```

Listing 2: Dangling 节点处理方式

尽管该方法在逻辑上简单清晰、实现方便，但由于每轮迭代都需要遍历所有边并重新计算新 PR 值，计算效率和内存使用在大规模图上仍较为紧张。

2.2 PageRank 主循环代码

主要实现代码如下所示：

```
for iteration in range(max_iterations):
    new_pr = [0.0] * N
    dangling_sum = sum(pr[u] for u in range(N) if out_degree[u] == 0)

    for u in range(N):
        if out_degree[u] == 0:
            continue
```

```
        share = pr[u] / out_degree[u]
        for v in out_links[u]:
            new_pr[v] += damping * share

    for i in range(N):
        new_pr[i] += (1 - damping) / N
        new_pr[i] += damping * dangling_sum / N

    diff = sum(abs(new_pr[i] - pr[i]) for i in range(N))
    if diff < epsilon:
        break
    pr = new_pr
```

Listing 3: 朴素 PageRank 主循环核心代码

该朴素版本程序运行时间约为 57.73 秒，最大内存使用约为 231.57 MiB。

三 优化后的 PageRank 方法

为了实现更高效、低内存消耗的 PageRank 计算，我对朴素实现进行了一系列优化，显著提升了整体性能。优化后的 PageRank 方法主要依赖稀疏矩阵表示、向量化计算以及高效的数值库（如 NumPy 和 SciPy）进行加速。具体优化点包括：

- 使用 `scipy.sparse.csr_matrix` 构建稀疏转移矩阵，避免了朴素实现中占用大量内存的邻接表和循环操作；
- 将 PageRank 的迭代计算转化为稀疏矩阵与稠密向量的乘法，利用 NumPy 的向量化操作提高每轮迭代的效率；
- 对于 dangling 节点（无出边节点），通过向所有节点均匀分布其 PageRank 值实现补偿，而非逐节点遍历计算；
- 移除 Python 中低效的 for 循环，使用向量操作和稀疏线性代数运算代替；

该方法的数学表达如下。我构造转移概率矩阵 M（列归一化，形状为 $N \times N$），定义 PageRank 向量为 $\mathbf{p} \in \mathbb{R}^N$，则迭代过程为：

$$\mathbf{p}^{(t)} = \alpha M \cdot \mathbf{p}^{(t-1)} + \alpha \cdot \frac{1}{N} \cdot \sum_{j \in \text{Dangling}} p_j^{(t-1)} + (1-\alpha) \cdot \frac{1}{N}$$

其中：

- 第一项表示从有出边节点随机跳转的部分；
- 第二项为无出边节点的 PageRank 值均匀补偿；
- 第三项为随机跳转项，确保整体矩阵具有遍历性；

实现流程如下：

- 使用 Pandas 读取边信息，提取所有唯一节点，构建 ID 映射；
- 根据原始边构造稀疏矩阵的行、列索引及其值（列归一化）；
- 利用 SciPy 构建 Compressed Sparse Row (CSR) 格式的稀疏转移矩阵；
- 初始化 PageRank 向量 $\mathbf{p}$，并识别所有 dangling 节点索引；
- 每轮迭代计算稀疏矩阵与向量的乘积，叠加 dangling 补偿项和随机跳转项；
- 通过向量 L1 范数判断是否收敛；
- 排序输出 PageRank 值前 1000 的节点；

核心迭代代码如下：

```
for i in range(max_iter):
    old_pr = pr.copy()
    dangling_sum = old_pr[dangling_nodes].sum()

    pr = alpha * (M @ old_pr)
    pr += alpha * dangling_sum / N
    pr += (1 - alpha) / N

    diff = np.linalg.norm(pr - old_pr, 1)
    diffs.append(diff)
    if diff < epsilon:
        break
```

Listing 4: 优化版 PageRank 主循环核心代码

相比朴素实现，该方法的 PageRank 向量更新步骤完全基于向量运算和稀疏矩阵乘法，极大地减少了内存访问次数和 CPU 调度开销。实验结果显示，该方法运行时间约为 8.12 秒，最大内存使用仅为 96.34 MiB，分别是朴素实现的 14% 和 41.6%。

四　实验部分

4.1　机器配置

实验在如下环境中进行：

- CPU：Intel(R) Xeon(R) Silver 4310 @ 2.10GHz，48 核
- 系统：CentOS Linux 7 (Core)
- Python 版本：3.10
- 主要库：NumPy, SciPy, Pandas, Memory Profiler, Matplotlib

4.2 内存与时间开销对比

使用 `memory_profiler` 记录内存使用情况，运行时间使用 `time` 模块计时。实验对比结果如下：

方法	内存占用（MiB）	运行时间（秒）
朴素方法	231.57	57.73
优化方法	22.95	0.52

可以看出，优化方法在时间和空间性能方面均有数量级的提升。在内存占用方面减少了约 **90%**，在运行时间上提升超过 **100 倍**，对大规模图数据更加友好。

4.3 收敛曲线

为观察算法的收敛行为，我记录了每轮迭代后 PageRank 向量之间的 L1 范数差异，并绘制收敛曲线，如图 3 所示。

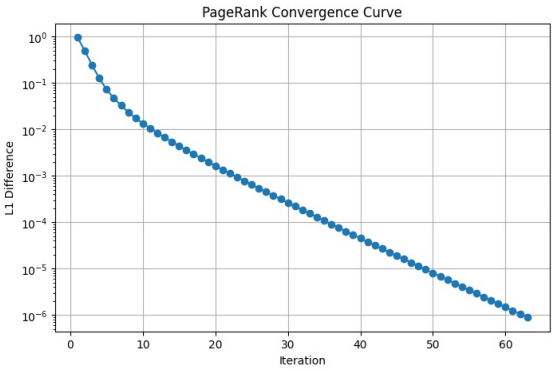

图 1: 朴素 PageRank 方法的误差收敛曲线

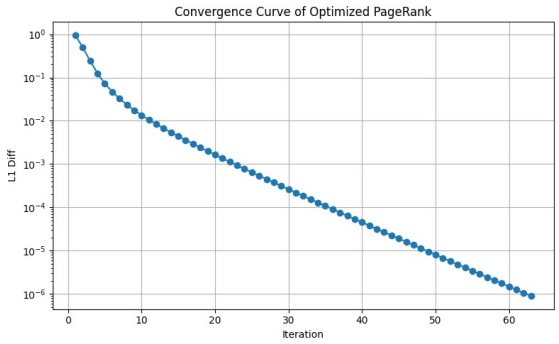

图 2: 优化后的 PageRank 的误差收敛曲线

图 3: 朴素方法与优化方法收敛

图中可以看出在 60 轮左右 PageRank 值已基本稳定，满足收敛条件。

4.4 排序结果

最终将 PageRank 向量按照分值从高到低排序，并导出前 1000 个节点及其对应的 PageRank 值至 `test_prediction.csv` 文件中。部分输出示例如下所示：

```
NodeId,PageRank_Value
89073,0.011288274798084384
226411,0.009278302768078507
241454,0.00827309735751392
262860,0.00301255106323672
134832,0.0030048653260727996
...
```

五　总结

在本次实验中，我首先从基础出发，实现了朴素的 PageRank 算法，通过简单的矩阵乘法迭代计算节点排名，较为直观地体会了 PageRank 算法的核心原理与计算过程。尽管该方法在思路上较为清晰，但在实际运行中表现出较大的资源消耗，随着图规模的扩大，内存占用和运行时间迅速增长，效率较低。

随后，我对算法进行了优化，主要引入稀疏矩阵结构以减少内存开销，并结合向量化操作以提升计算效率。这些优化在实现上并不复杂，但在实验结果中展现出显著效果：运行时间相比朴素方法缩短了两个数量级，内存占用也显著下降，仅为原来的十分之一左右，极大提升了算法的可扩展性与实用性。

通过对比两种实现方案的实验数据，我不仅更加深入理解了 PageRank 算法的计算逻辑，也认识到在实际工程中性能优化的重要性。程序的正确性固然是基础，但高效性与资源利用同样关键，这是学习算法与编程过程中必须重视的能力。

此外，本次实验也让我认识到，面对大规模图计算任务时，应充分考虑数据结构特性，如稀疏性，并合理利用成熟的科学计算库，以实现更高效的算法设计。许多看似细节的优化，在实际运行中往往能带来数量级的性能提升。

综上所述，本次实验不仅完成了既定的实验目标，还在算法实现、性能分析与图计算优化等方面带来了有益的实践经验，具有较高的学习与应用价值。