南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

大数据计算及应用期中实验报告

Compute the PageRank scores on the given dataset

年级：2020 级

小组成员：许克婧 2013211
管昀玫 2013750
黄丹禹 2012030

指导教师：杨征路

2023 年 4 月 30 日

摘要

本次实验研究 Pagerank 算法及其改进方法在网页重要性排序中的应用，以及通过实现和比较基础方法和 block stripe 方法来探索其性能和效率的提升。我们首先实现了基本的 Pagerank 算法，然后为解决实际应用中内存不足的问题，实现了引入了基于 block stripe 分块的 Pagerank 算法。此外，我们通过调用库函数生成了标准结果，并将 basic 版本和使用 block stripe 优化的结果与标准结果进行了对比。本次实验为理解和应用 Pagerank 算法提供了基础，同时为今后更深入的研究和改进提供了启示。

关键字：Pagerank, Block-Stripe

目录

一、 数据集说明

本次实验的数据集为 data.txt，其数据格式为每行按照 <srcNodeID, dstNodeID> 的方式进行存储，表示图中存在从结点 srcNodeID 到结点 dstNodeID 的一条边。

据统计我们知道共有 83852 条数据，但其中存在着 2100 条重复数据记录，所以在载入数据的时候注意对数据记录进行去重处理，以免造成统计上的错误，给 PageRank 分值的计算带来不必要的误差（如图1），具体实现见后面部分：

```
PS D:\LessonProjects\Big-Data> & D:/Python39/python.exe d:/LessonProjects/Big-Data/ReFactor/PageRank.py
This is Basic Page Rank
..\Results is exist.
Middle is exist.
num of total links: 83852
num of unique links: 81752
max node number: 8297
read_graph:  102.00s
number of unique nodes:
Iter: 0, Before adjusting, S value: 0.74590451860130935
```

图 1: Unique Links Number

数据集中最大结点编号为 8297，但是注意并非所有结点都有出入度，存在出度和入度都为 0 的孤立结点，所以图中实际需要排名的结点数 N 少于结点最大编号值（本实验中 N = 6263，如图2），代码实现时应注意删除这种孤立结点，否则会给排名结果带来很大的误差，具体实现见第二部分3。

```
max node number: 8297
read_graph:  137.65s
number of unique nodes:6263
```

图 2: Unique Nodes Number

除此之外，有向图中存在只有入度但无出度的结点，即 dead end，一旦跳转到此类结点，则无法继续进行跳转。图中还存在没有出度的结点集，即 spider trap，一旦跳转到这样的结点集当中，就只能继续在该结点集内部进行跳转而不能跳转到其以外的其他结点。这两种结点都会使 PageRank 值计算错误。在实验中我们通过加入随机跳转因子 telesport 以及修正 PageRank 排名列向量来减少这两类结点造成的负面影响，具体实现见第二部分详细实现2。

二、 PageRank

(一) 算法概述

1. 基本思想

在实际应用中许多数据都以图（graph）的形式存在，比如，互联网、社交网络都可以看作是一个图。图数据上的机器学习具有理论与应用上的重要意义。PageRank 算法是图的链接分析（link analysis）的代表性算法，属于图数据上的无监督学习方法。

PageRank 算法是谷歌搜索引擎的核心算法之一，它通过分析网页间的链接关系来确定每个网页的重要性，并将结果用于搜索结果排序。[3] [2]

PageRank 算法的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。[1]

PageRank 算法中每个节点的得分由以下两部分组成：

1. 其他网页指向该网页的链接数目，即入度；
2. 其他网页的 PageRank 值与它们的出度之和的乘积之和，即出度。

PageRank 算法的核心是通过反复迭代计算每个节点的 PageRank 值，直到收敛。迭代计算的过程中，每个节点的 PageRank 值会随着时间的推移而逐渐趋于稳定值，即网页的“重要性”。

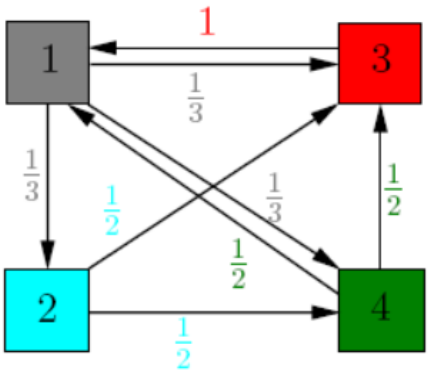
2. 算法描述

PageRank 算法的计算流程可以分为以下几个步骤：

1. 初始化所有网页的 PageRank 值和为 1.0，每个网页的 PageRank 值为 1/N；
2. 根据网页之间的链接关系计算出每个网页的出度；
3. 迭代计算每个网页的 PageRank 值，直到达到一定的收敛条件；
4. 对所有网页的 PageRank 值进行归一化处理，使得它们的和等于 1

其中，收敛条件通常是设置一个误差阈值，当两次迭代之间所有节点的 PageRank 值差的绝对值均小于该误差阈值时，算法停止迭代。

具体迭代过程，我们举例来说明：



Let us denote by A the transition matrix of the graph, $A = \begin{bmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{bmatrix}$.

图 3: Example, from http://pi.math.cornell.edu/

每个页面都应该将其重要性均匀地转移到它链接到的页面上。节点 1 有 3 个输出边，因此它将其重要性传递给其他 3 个节点中的每一个。节点 3 只有一个输出边，因此它将把所有重要性传递给节点 1。一般来说，如果一个节点有 k 个外向边，它会将其重要性传递给它链接到的每个节点。

使用 v 来代表初始秩向量。使用 v 不断右乘转移矩阵 A 直到收敛条件，即可得到我们需要的 PageRank 向量，如图4所示。

$$\mathbf{v} = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}, \quad \mathbf{Av} = \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix}, \quad \mathbf{A^2 v} = \mathbf{A(Av)} = \mathbf{A} \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix} = \begin{pmatrix} 0.43 \\ 0.12 \\ 0.27 \\ 0.16 \end{pmatrix}$$

$$\mathbf{A^3 v} = \begin{pmatrix} 0.35 \\ 0.14 \\ 0.29 \\ 0.20 \end{pmatrix}, \quad \mathbf{A^4 v} = \begin{pmatrix} 0.39 \\ 0.11 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad \mathbf{A^5 v} = \begin{pmatrix} 0.39 \\ 0.13 \\ 0.28 \\ 0.19 \end{pmatrix}$$

$$\mathbf{A^6 v} = \begin{pmatrix} 0.38 \\ 0.13 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad \mathbf{A^7 v} = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}, \quad \mathbf{A^8 v} = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}$$

图 4: 计算过程, from http://pi.math.cornell.edu/

迭代序列 $v, Av, ..., A^k v$ 最终趋向平衡值 v^*，而 v^* 就是最终的 PageRank 向量。

PageRank 算法的实现需要考虑以下几个细节：

1. 在计算网页出度时，需要考虑网页指向自身的链接（自环）的情况；
2. 为避免图中存在孤立节点（即没有任何入度或出度的节点），可以采用随机跳转技术，在每次迭代时，以一定的概率随机跳转到图中的任意节点。

为了解决这两个模型，拉里·佩奇提出了 PageRank 的随机浏览模型。他假设了这样一个场景：用户并不都是按照跳转链接的方式来上网，还有一种可能是不论当前处于哪个页面，都有概率访问到其他任意的页面，比如说用户就是要直接输入网址访问其他页面，虽然这个概率比较小。

所以他定义了阻尼因子 d，这个因子代表了用户按照跳转链接来上网的概率，通常可以取一个固定值 0.85，而 1-d=0.15 则代表了用户不是通过跳转链接的方式来访问网页的，比如直接输入网址。

因此，最终 PageRank 的公式为：

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

其中 N 为网页总数。至此，我们又可以重新迭代网页的权重计算了，因为加入了阻尼因子 d，一定程度上解决了等级泄露和等级沉没的问题。

数学定理已证明，最终 PageRank 随机能够收敛。

算法的伪代码如下图所示：

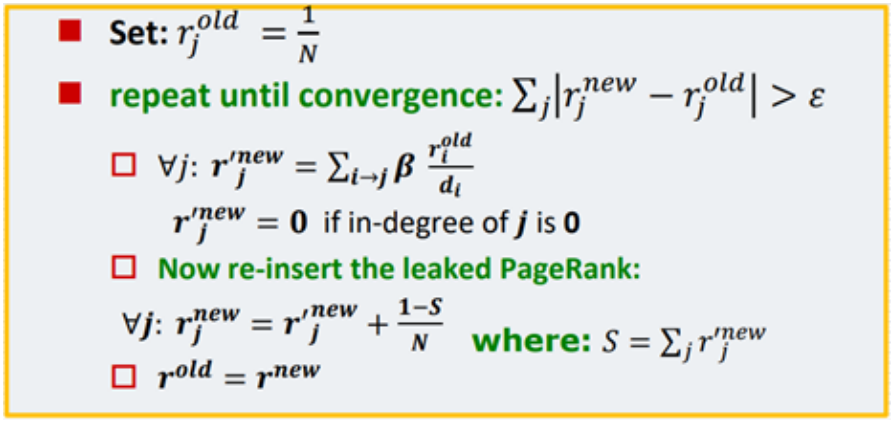


图 5: 基本 PageRank 算法（摘自课堂 PPT）

(二) 具体实现

1. 构建图并载入数据

使用一个入度形式的邻接链表 G 存储图像信息，G_{ij} 表示第 j 个节点指向第 i 个节点。在预处理时我们已经得到节点总数 N 和最大节点编号 max_node（注意，N 不为最大节点数），为了计算方便，可以使用长度为 max_node 的数组来表示 G，使用长度可变的无序链表来表示 G_i。但同时计算时需要注意让那些并不存在的节点 PageRank 值始终为 0。

此外，还需要一个长度为 N 的的出度表 D，D_i 表示第 i 个节点的出度。数组和无序链表在 python 实现中可均使用 list。G 和 D 可以在一次遍历数据集过程中载入完成，因此该过程时间复杂度为 $O(M)$，需要的内存约为 $(2N + M)B$ 字节。

由于数据集中存在重复数据，影响实验结果，故我们从文件中逐行读取后，首先去除重复的记录得到 nodes 列表。然后提取出最大结点序号和所有边，构建 Graph 类型的有向图 g，具体实现如下：

Class Graph

```
class Graph:
    """
    Using lil matrix: src, in-edges, out-degree
    """
    def __init__(self, nnodes, edges) -> None:
        self.nnodes = nnodes  # number of nodes
        self.in_edges = defaultdict(list) # in-edge list
        self.out_degrees = [0] * (nnodes + 1)

        for i, (src, dst) in enumerate(edges):
            self.out_degrees[src] += 1 # update out-degree
            self.in_edges[dst].append(src) # add in-edge
```

通过其中的 in_edges 和 out_degrees 列表承载图中所有结点的出入度信息，out_degrees[i] 代表 i 结点的出度数量，in_edges[i] 代表 i 结点的入度结点列表，其中 in_edges[i][j](j=0,1,2⋯) 依次为每个指向 i 节点的结点序号。

建立图类对象时首先读取数据并剔除其中的重复记录，载入图数据的代码如下：

load graph

```python
def load_graph() -> Graph:
    # Read data from dataset
    links = []
    with open("Data\data.txt", "r", encoding="utf-8") as file:
        for line in file:
            src, dst = map(int, line.split())
            links.append((src, dst))

        # Get unique list of nodes
        nodes=[]
        for row in links:
            if row not in nodes:
                nodes.append(row)
        print("num of total links: {}".format(len(links)))
        print("num of unique links: {}".format(len(nodes)))

        # Get unique list of nodes
        max_both = -1 # max number of nodes (maybe != total number of nodes)
        for i in links:
            max_both = max(max_both, i[0], i[1])

        print("max node number: {}".format(max_both))
        return Graph(max_both, nodes)
```

2. 处理 dead ends & spider trap

在 PageRank 算法中，Dead Ends 和 Spider Traps 是两种可能会导致算法不收敛或结果偏离实际情况的问题。Dead Ends 是指在图中存在没有出度的节点，这些节点在 PageRank 算法中的贡献会被浪费掉，可能会导致算法不收敛。Spider Traps 是指在图中存在一个或多个由相互指向的节点组成的环，这些环可能会导致算法收敛到一个错误的结果上。

处理这两个问题即使用随机游走模型。即在每次迭代时，以一定的概率随机跳转到图中的任意节点，这样可以保证算法始终能够遍历整个图，从而避免 Dead Ends 和 Spider Trap 的问题。在这里，这个概率即为超参数 $1 - TELEPORT$，$TELEPORT$ 依据经验被设定为 0.85。在本实验的第三部分将会对这个超参数进行进一步讨论。

Dead End and Spider Trap

```python
    # Note: idx=0 is useless, we set it for convinience
    r_new = np.zeros(graph.nnodes + 1, dtype=np.float64)
    for dst in range(graph.nnodes+ 1):
        if len(graph.in_edges[dst]) == 0:  # in-degree = 0
            r_new[dst] = 0
        else:
            for src in graph.in_edges[dst]:
                # $$ r*{new}_j = \sum_{i\rightarrow j}\beta *
                # frac{r*{old}_i}{d_i} $$
                # Note: \beta = TELEPORT. TELEPORT = 0.85
```

```
                r_new[dst] += (r_old[src] \
                    / graph.out_degrees[src])*TELEPORT
    s = np.float64(np.sum(r_new))
    print("Iter: {}, Before adjusting, S value: {:.17f}".format(iter, s))
    r_new += (1 - s) / N # Now re-insert the leaked PageRank
    r_new[is_isolate] = 0
    s = np.float64(np.sum(r_new))
    print("Iter: {}, After adjusting, S value: {:.17f}".format(iter, s))
```

3. 孤立结点的处理

在 pagerank 算法随机游走的过程中，需要以一固定概率跳转到其他页面上。但在对数据集的分析中我们发现，数据集中出现的最小序号结点为 3，最大序号节结点为 8273，但出现的结点个数只有 6263 个，说明有一些节点没有入度和出度，所以对于这些“孤立结点”，我们并不将其列入跳转的目的结点范畴。由于在随机游走中的概率分布不应该包含这些结点，所以在 r_new 和 r_old 数组中，需要将这些结点的 pagerank 值设置为 0，并调整 $\frac{1-S}{N}$ 中 N 的大小为“非孤立结点”个数。具体实现如下，首先统计出现的结点总数 N，其次保持孤立结点的 pagerank 值在计算过程中始终为 0:

Listing 1: 统计出现的结点总数 N

```
    i = 0
    count = 0
    is_isolate = np.full(graph.nnodes + 1, True)#用于记录孤立节点
    while i < (graph.nnodes + 1):
        #若出度为0且入度为0则排除
        if not (graph.out_degrees[i] == 0 and \
                len(graph.in_edges[i]) == 0):  # in- and out-degree = 0
            count += 1
            is_isolate[i] = False
        i += 1
    N = count
    print("实际节点数:%d "%N)
```

Listing 2: 初始化——只初始化非孤立结点的 pagerank 值为 1/N

```
    #初始化pagerank值矩阵为1/N
    r_old = np.full(graph.nnodes + 1, np.float64(1 / N))
    #保持孤立节点的pagerank值为0
    r_old[is_isolate] = 0
```

Listing 3: 进行修正并保持孤立结点 pagerank 值为 0

```
        s = np.float64(np.sum(r_new))
        # s = np.sum(r_new)
        print("Iter: {}, Before adjusting, S value: {:.17f}".format(iter, s))
        r_new += (1 - s) / N  # Now re-insert the leaked PageRank
        r_new[is_isolate] = 0#保持孤立结点pagerank值为0方可不影响计算
        s = np.float64(np.sum(r_new))
```

```
7         print("Iter: {}, After adjusting, S value: {:.17f}".format(iter, s))
```

4. 计算 pagerank 值

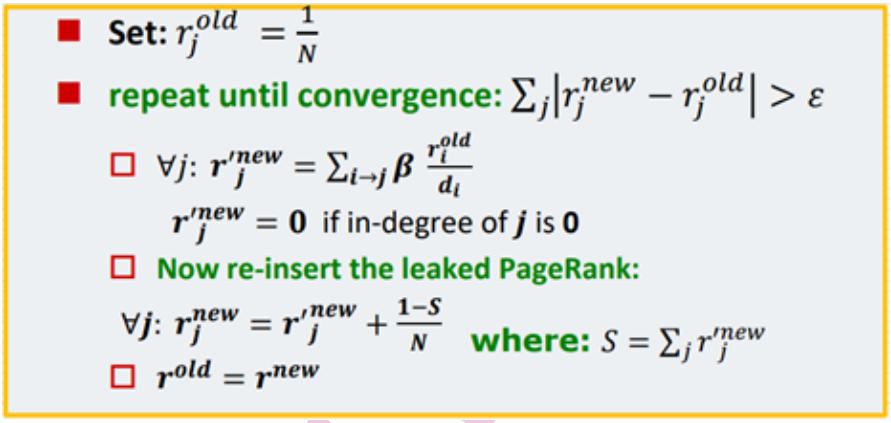
算法细节已由上述叙述给出，接下来阐述整体计算流程和整体 PageRank 计算代码：

该过程额外需要维护两个长度为 N 的数组 r_{new} 和 r_{old}。得益于 G 的数据结构，指向节点 j 的节点 i 存储在连续的链表上并且能够以 $O(1)$ 的时间定位到链表的起点，同样的能以 $O(1)$ 的时间获得第 i 个节点的出度 d_i。在使用 python 实现的过程中，可以使用已经加速的 numpy 库实现向量求和，向量与标量的加法等操作。

使用 r_{new} 和 r_old 的 L1 距离或者 L2 距离作为绝对误差，当绝对误差小于超参数 ϵ 或迭代次数超过超参数 MAX_ITER 时，算法终止。该过程的时间复杂度为 $O(I(M+N))$，需要的内存约为 $2N \times B$ 字节。

结果输出需要将 r_{new} 按照得分由大到小排序，将前 100 名的结果写入文件，时间复杂度为 $O(NlogN)$。

基本算法完全运行在内存上，总共需要的内存为 $(4N+M)B$，会随着节点数和边数线性增长。在实际场景下，由于节点和边的数量非常庞大，该算法有内存不足的风险。




```python
def pagerank(graph: Graph):
    # Preprocessing
    # While calculating N, in-degree = 0 is not considered.
    # However, the lil matrix is still the original graph.
    i = 0
    count = 0
    is_isolate = np.full(graph.nnodes+1, True)
    while i < (graph.nnodes+1):
        if not (graph.out_degrees[i] == 0 and \
                len(graph.in_edges[i])==0): # in- or out-degree = 0
            count += 1
            is_isolate[i] = False
        i += 1
    N = count
    print("number of unique nodes:{} ".format(count))

    # Initializing matrix
    r_old = np.full(graph.nnodes+1, np.float64(1 / N))
    # Note: isolate node is not considered while calculating r_old and r_new
    r_old[is_isolate] = 0

```

```
    #iteration
    iter = 0
    while True:
        # Note: idx=0 is useless, we set it for convinience
        r_new = np.zeros(graph.nnodes + 1, dtype=np.float64)
        for dst in range(graph.nnodes+ 1):
            if len(graph.in_edges[dst]) == 0:  # in-degree = 0
                r_new[dst] = 0
            else:
                for src in graph.in_edges[dst]:
                    # $$ r*{new}_j = \sum_{i\rightarrow j}\beta *
                    # frac{r*{old}_i}{d_i} $$
                    # Note: \beta = TELEPORT. TELEPORT = 0.85
                    r_new[dst] += (r_old[src] \
                        / graph.out_degrees[src])*TELEPORT
        s = np.float64(np.sum(r_new))
        print("Iter: {}, Before adjusting, S value: {:.17f}".format(iter, s))
        r_new += (1 - s) / N # Now re-insert the leaked PageRank
        r_new[is_isolate] = 0
        s = np.float64(np.sum(r_new))
        print("Iter: {}, After adjusting, S value: {:.17f}".format(iter, s))

        iter += 1

        # Find max error
        error = 0.
        for i in range(graph.nnodes + 1):
            if(error < abs(r_new[i] - r_old[i])):
                error = abs(r_new[i] - r_old[i])

        if (error < EPSILON or iter >= MAX_ITER):
            print(f"absolute error: {error}, iter: {iter}")
            break
        r_old = copy.deepcopy(r_new)

    result = {}
    for i in range(1, graph.nnodes+1):
        result[i] = r_new[i]

    return result
```

5. 收敛条件

经多次实验，我们终于找到了最佳的收敛条件，如下：

在每一次迭代中，我们选取 pagerank 值前后变化最大的结点作为对象（多次迭代过程中，每次判断的结点可能不同），当 PageRank 值变化最大的结点在本次迭代中绝对误差小于阈值 $EPSILON$，那么其他结点的绝对误差必然已小于阈值，代表已经收敛；或达到最大迭代次数

（本实验为 100），完成迭代。具体判断如下：

```
    error=0
    # 寻找本次迭代中pagerank值变化最大的结点i作为判断对象
    for i in range(graph.nnodes+1):
        if(error<abs(r_new[i]-r_old[i])):
            error=abs(r_new[i]-r_old[i])
    # 1. 若pagerank值变化最大的结点绝对误差小于阈值代表收敛
    # 2. 达到最大迭代次数完成迭代
    if(error<EPSILON or iter>=MAX_ITER):
        print(f"absolute error: {error}, iter: {iter}")
        break
```

在实验中，我们多次调整阈值 EPSILON 的大小，以平衡算法的精确性和速度，最终得到结果——当阈值 EPSILON 为 0.0001 时，与调库结果相似度最高。

三、 Block-stripe

(一) 算法概述

在现实情景下，PageRank 算法的性能受到存储介质空间的限制。内存空间，甚至单台机器的硬盘存储空间，难以支持庞大的数据以及 PageRank 的复杂运算，因此，分块化的 PageRank 算法优化显得格外重要。

为了解决存储空间有限的问题，可以采用分块的思想。即把进行解耦合并分块处理。每个块内部先进行一次处理，之后把处理之后的结果，通过整合、处理，得到最终的结果。对于每一块，通常认为是单台计算机可以存储并且处理的，而最终的整合信息，也认为是单台计算机可以综合处理的。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。

工程领域中，常采用 MapReduce 分布式集群的模型架构。由于实验要求，本次实验采取 Block Stripe 方法。Block Stripe 的 PageRank 算法主要由以下几个步骤组成：

1. 将整个网页图分成若干个 Block，并为每个 Block 分配一个标识符。可以按照某种规则（如按照节点 ID 或按照节点出度等）对节点进行划分。

2. 将每个 Block 中的节点信息（包括每个节点的 ID、出度以及指向其他节点的链接）分配给不同的机器或处理器进行计算。每个机器或处理器只需要处理自己负责的 Block 中的节点信息，不需要关心其他 Block 中的节点信息。

3. 在计算每个节点的 PageRank 值时，需要考虑来自其他 Block 中节点的链接。可以通过在每个 Block 中维护一个出度矩阵和一个入度矩阵来实现。出度矩阵记录每个节点指向其他节点的链接，入度矩阵记录每个节点被其他节点指向的链接。

4. 每个机器或处理器在计算完自己负责的 Block 中的节点 PageRank 值后，需要将结果发送给其他机器或处理器进行合并。可以采用 MPI（Message Passing Interface）等通信协议实现机器间的通信和数据交换。在本次实验中，为了简化，我们采取写入文件的格式。

5. 最后，将所有 Block 的 PageRank 值合并起来，得到整个网页图的 PageRank 值。

(二) 具体实现

在本实验中，基于上述思想，我们完成了基于 Block-Stripe 的 PangeRank 算法实现：假设磁盘能容纳所有数据，考虑将 r_new 分成 k 块，每块的大小为 compute_size，每次计算只读取 1 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，在计算时从磁盘加载所需块，大大减少了计算所需内存。

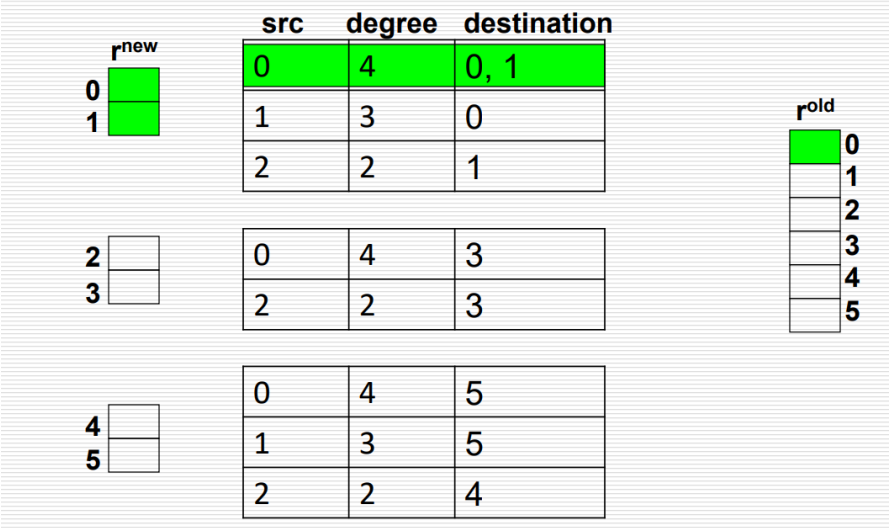


图 6: Block-stripe 分块（摘自课堂 PPT）

1. 数据集预处理

根据数据集特点对其进行预处理操作。

首先遍历读取文件，并剔除重复数据记录。读取过程复杂度为 $O(N)$，其中 N 为链接的数量。

```
edges = []
with open(data_txt, 'r', encoding="utf-16") as f:
    for line in f:
        src, dest = map(int, line.strip().split())
        if (src, dest) not in edges:  # 重复数据去重
            edges.append((src, dest))
```

由于 Block Stripe 的性质，很难让我们像 naive PageRank 一样去记录并不存在的节点，并在计算过程中把他们的 PageRank 值设置为 0，因此我们选择在预处理部分直接将原节点编号映射为新节点编号。即：从 nodeID 1 到 nodeID N，序号顺序排列，中间无空缺值。nodeID N 是最大节点编号，也是总节点数量。

预处理后的数据存储在 new_data.txt 中。

mapping

```
# 读取文件并建立节点编号映射关系
nodes = set()
for edge in edges:  # 获取实际结点个数
    nodes.add(edge[0])
    nodes.add(edge[1])
```

```
    node_list = sorted(list(nodes))  # 得到排序后的节点列表

    # 建立节点映射字典，将旧id映射为新id
    node_dict = {}
    for i, node_id in enumerate(node_list):
        node_dict[node_id] = i + 1
    # 将边列表中的节点id映射为新id
    new_edges = []
    for edge in edges:
        new_src = node_dict[edge[0]]
        new_dest = node_dict[edge[1]]
        new_edges.append((new_src, new_dest))

    with open(new_data_txt, 'w') as f:
        for edge in new_edges:
            new_src = edge[0]
            new_dest = edge[1]
            f.write(str(new_src) + " " + str(new_dest) + "\n")
```

注意，在 PageRank 值计算结束后，需要再将新的 nodeID 映射回原来的 nodeID，方可得到正确的结果输出。

remap

```
    # 将映射字典键值反转，将结点编号映射回去
        reverse_mapping = {v: k for k, v in node_dict.items()}
        result_original = {}
        for mapped_node, pagerank in result.items():
            original_node = reverse_mapping[mapped_node]
            result_original[original_node] = pagerank
```

我们现在做出一个相对较弱的假设：假设内存中能够存放 block_size 条记录，其中每一条记录都是一个字典，字典的 key 为 *dst* 节点，value 为所有的 *src* 节点形成的一个 list。利用 python 读数据是按照行进行读取的特点，我们可以在一遍扫描的过程中就形成所述的 block_size 条记录，并实现分块存储。

具体实现为：在每次读取一条边时，判断 *dst* 是否已经在当前记录中，如果没有则新添加一条记录；如果有则判断这条边是否已经存在，不存在则添加，存在则不添加。当记录的条数达到预设的 block_size 时，将当前所有的记录写磁盘保存，并清空内存。在执行上述的过程中，还会维持一个字典，字典的 key 为 *dst* 节点，value 为其保存的文件序号，这样有利于我们后续形成计算所用分块时的定位。

我们利用倒排索引来帮助我们完成这一部分的构建：

remap

```
def pre_block(data):
    """
        :param data: path of data.txt
        :return: None
        for each element in matrix: src:[degree, [list_of_out]]
```

```
    """
    print("Preprocessing Blocks")

    pages = -1
    name_count = 0
    # 使用defaultdict替换字典的setdefault方法，可以更简洁地初始化字典中的值
    s_matrix = defaultdict(list)
    out_of_nodes = defaultdict(int)
    files_of_des = defaultdict(list)
    current_num_des_nodes = []
    # 建立倒排索引
    with open(data, 'r', encoding='utf-8') as f:
        for line in f:
            src = int(line.split()[0].split()[0], 10)
            des = int(line.split()[1].split()[0], 10)
            pages = max(src, pages, des)
            # 记录出度
            out_of_nodes[src] += 1

            # 记录des所在的文件
            if name_count not in files_of_des[des]:
                files_of_des[des].append(name_count)

            if des not in current_num_des_nodes:
                if len(current_num_des_nodes) % block_size == 0:
                    if len(current_num_des_nodes) != 0:
                        # 如果此时长度达到要求，需要保存内容
                        # 保存当前list里的所有内容到磁盘中
                        save_data_for_RI(s_matrix, name_count)
                        name_count += 1

                        # 清空内存
                        s_matrix = defaultdict(list)
                        current_num_des_nodes = []

                        # 保存新的内容
                        s_matrix[des].append(src)
                        current_num_des_nodes.append(des)

                    else:
                        # curr为0，表示没有内容，直接保存新的内容到内存中即可
                        s_matrix[des].append(src)
                        current_num_des_nodes.append(des)
                else:
                    # des不在列表里，此时长度没有到达要求，直接保存到内存中
                    s_matrix[des].append(src)
                    current_num_des_nodes.append(des)
            else:
```

```
                # 已有des, 保存即可
                s_matrix[des].append(src)
        # 处理最后一个块
        save_data_for_RI(s_matrix, name_count)

        with open(pages_txt, 'w', encoding='utf-8') as pages_file:
            pages_file.write(str(pages))
        # 清空内存
        s_matrix = defaultdict(list)
        current_num_des_nodes = []

    print("RevertedIndex Data Saved")
    process_block_data(pages, compute_size, files_of_des, out_of_nodes)
```

其中，保存倒排索引文件的函数为:

Save RI

```
def save_data_for_RI(s_matrix, name_count):
    # 排序
    s_matrix = dict(sorted(s_matrix.items(), key=lambda d: d[0], reverse=
        False))
    for item_des in s_matrix.keys():
        s_matrix[item_des].sort()

    # 保存当前list里的所有内容到磁盘中
    reverse_file_name = Directly_REVERSE_MATRIX_FOLDER + 'Directly_reverse_'
        + str(name_count) + '.txt'
    with open(reverse_file_name, 'w', encoding='utf-8') as save_f:
        save_f.write(json.dumps(s_matrix))
```

2. 建立计算分块

将 r_new 分成 k 块，每块的大小为 compute_size，所以更新 r_{new} 中的 compute _size 个值。由此我们在分块的时候将含有有目标更新 compute _size 个 dst 的边聚合成一个个分块，存放在稀疏矩阵里，稀疏矩阵的格式为 $src : [degree, [dst]]$, 这些分块作为分块文件按序存到磁盘上，按需读取。建立分块时，由于我们已经将数据集预处理重整其结构为 dst: [src]，所以能快速查找目标结点 dst 所在稀疏矩阵中的位置从而将矩阵分块, 不赘述。分块具体处理函数为 process_block_data(), 此处列举部分代码:

```
def process_block_data(pages, compute_size, files_of_des, out_of_nodes):
    # save file according to the value of size
    k = math.ceil(pages / compute_size)

    for num in range(k):
        save_file_name = os.path.join(SAVE_DATA_FOLDER, f'{num}.txt')
        temp_save_dict = dict()
        temp_count = 1
        while (temp_count % compute_size != 1 or temp_count == 1):
```

```
            des = temp_count + num * compute_size
            temp_count += 1

            # if reverse_matrix.get(des) is None : continue
            if des not in list(files_of_des.keys()): continue

            # 定位在哪一个reverse文件里
            des_src_list = []
            for file_number in files_of_des[des]:
                des_reverse_file = Directly_REVERSE_MATRIX_FOLDER + '
                    Directly_reverse_' + str(file_number) + '.txt'

                with open(des_reverse_file, 'r', encoding='utf-8') as f:
                    reverse_data = json.loads(f.read())
                    if not reverse_data.get(str(des)) is None:
                        des_src_list.extend(reverse_data[str(des)])

            for src in des_src_list:
                if temp_save_dict.get(src) is None:
                    # 定位src的d
                    d = out_of_nodes[src]
                    temp_save_dict[src] = [d, [des]]
                else:
                    temp_save_dict[src][1].append(des)

        # 排序并保存
        temp_save_dict = dict(sorted(temp_save_dict.items(), key=lambda d: d
            [0], reverse=False))

        for key in temp_save_dict:
            temp_save_dict[key][1].sort()

        with open(save_file_name, 'w', encoding='utf-8') as out:
            out.write(json.dumps(temp_save_dict))

        if (temp_count + num * compute_size) > pages: break
    print("Block Data Saved")
```

3. 计算 PageRank 值

更新时，依次读取 compute_size 个 r_old 的值进行计算，将中间结果存储到 r_temp 中。对于读取到内存中的目标结点编号，到磁盘上读取相应的已经建立好的 Block-Stripe 分块，扫描稀疏矩阵，按照如下方法将 r_old 进行更新。

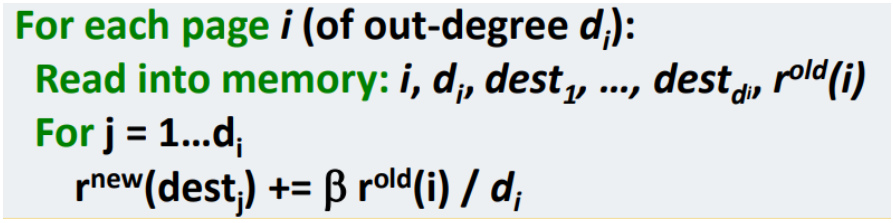


图 7: 一次迭代步骤（摘自课堂 PPT）

对于 dead end 和 spider trap 问题，同样采用基本 PR 算法中的方法处理：即每轮更新完毕后将结果写入到列向量 r_temp 中，最后读取 r_temp 中的值对其加和得到 S，计算 $1-S/N$ 将其加到 _temp 中作为修正并写出到 r_new 中。

收敛条件同基本 PageRank 算法一致。

稀疏矩阵相乘的代码定义在 sparse_matrix_multiply() 函数中, 如下:

```
def sparse_matrix_multiply(size , N, M):
    """
    compute one block each time, using one block of M and r_old
    :param size: size of each block
    :param N: number of dot
    :param M: the folder of M
    :param r_old: the folder of r_old
    :return: r_new, S
    r_new: a list of N/K elements, which K is the number of block
    S: current S
    """
    with open('./r_new.txt', 'w', encoding='utf-8') as new:
        new.truncate()
    with open('./r_new_temp.txt', 'w', encoding='utf-8') as new:
        new.truncate()
    with open('./r_old.txt', 'w', encoding='utf-8') as old:
        old.truncate()
    # 初始化 r_old
    temp = np.full(N, 1/N, dtype=np.float64)
    with open('./r_old.txt', 'w', encoding='utf-8') as t:
        for i in temp:
            t.write(str(i) + '\n')
    if_end = 0
    m_list = os.listdir(M)
    K = N // size
    if N % size != 0:
        K += 1
    print(K)
    E = 0.0001 #TODO originally E = 1e-9, maybe still needs modify

    while not if_end:
        r_new = np.zeros(size)
        start = 1  # 每一块r_new的起始索引
        end = start + size - 1  # 每一块r_new的结束索引: 每次只计算size次!
```

```
        S = 0

        for matrixs in range(len(m_list)):
            m_now = M + str(matrixs) + '.txt'
            with open(m_now, 'r', encoding='utf-8') as f1:
                matrix = json.loads(f1.read())#从磁盘读取文件进内存
            if matrix == {}:
                continue
            else:
                with open('./r_old.txt', 'r', encoding='utf-8') as f2:
                    count = 1
                    i = f2.readline().rstrip()
                    for src in matrix:
                        while int(src, 10) != count:
                            count += 1
                            i = f2.readline().rstrip()
                        if int(src, 10) == count:
                            d = matrix[src][0]
                            i = float(i)
                            for des in matrix[src][1]:
                                t = des
                                if t > end:
                                    c = t - start
                                    c = c // size * size
                                    with open('./r_new_temp.txt', 'a',
                                              encoding='utf-8') as f3:
                                        for x in range(c):
                                            f3.write('0.0\n')
                                    start += c
                                    end = start + size - 1
                                r_new[des - start] += b * i / d
                            count += 1
                            i = f2.readline().rstrip()
                        else:
                            count += 1

            start += size
            end = start + size - 1

            if end > N:
                end = N
            with open('./r_new_temp.txt', 'a', encoding='utf-8') as f3:
                for j in r_new:
                    f3.write(str(j) + '\n')
                    S += j
            if start <= end:
                r_new = np.zeros(end - start + 1)

```

```
82          e = 0.0
83
84          #进行修正
85          with open('./r_new_temp.txt', 'r', encoding='utf-8') as t:
86              with open('./r_new.txt', 'w', encoding='utf-8') as new:
87                  for i in t:
88                      i = float(i)
89                      new.write(str((i + (1 - S) / N)) + '\n')
90
91          #寻找最大误差
92          with open('./r_new.txt', 'r', encoding='utf-8') as new:
93              with open('./r_old.txt', 'r', encoding='utf-8') as old:
94                  for i, j in zip(new, old):
95                      # e += abs(float(i)-float(j))
96                      tmp = abs(float(i)-float(j))
97                      if e < tmp:
98                          e = tmp
99
100         #将old更新为new值
101         with open('./r_new.txt', 'r', encoding='utf-8') as new:
102             with open('./r_old.txt', 'w', encoding='utf-8') as old:
103                 old.truncate()
104                 for i in new:
105                     i = float(i)
106                     old.write(str(i) + '\n')
107
108         print(e)
109         if abs(e) <= E:
110             if_end = 1
111         else:
112             with open('./r_new.txt', 'w', encoding='utf-8') as new:
113                 new.truncate()
114             with open('./r_new_temp.txt', 'w', encoding='utf-8') as new:
115                 new.truncate()
```

四、 实验结果与结果分析

(一) 实验运行结果

1. 基础 pagerank 算法

经过测试, 本实验中设置 Random Teleports 值为 0.85; 误差采用绝对误差, 即前后 pagerank 值差的绝对值表示; 收敛阈值 EPSILON 为 0.0001; 最大迭代次数为 100。运行实验结果前十名如下:

结点排序	结点序号	pagerank 值
1	4037	0.004556412278710504
2	2625	0.00384595801934068
3	6634	0.0037219348234158885
4	15	0.00315532342605842
5	2398	0.0026710823701364757
6	2328	0.0026171312535371557
7	2470	0.0023842205688539907
8	5412	0.0023820335066259027
9	7632	0.0022811214618481553
10	3089	0.0022616729833260453

表 1: basic_result

2. Block-stripe

设置 Random Teleports 值为 0.85，误差采用绝对误差，即前后 pagerank 值差的绝对值表示；收敛阈值 EPSILON 为 0.0001；最大迭代次数为 100。r_new 被分成若干块，每块大小为 compute_size=50，稀疏矩阵因此相应地被分成 125 个 stipe。实验结果前十名如下:

结点排序	结点序号	pagerank 值
1	4037	0.004558981111374057
2	2625	0.0038478976725783384
3	6634	0.003723450379818571
4	15	0.003156950745872907
5	2398	0.0026723054306863406
6	2328	0.0026185555446699284
7	2470	0.0023856315444724953
8	5412	0.0023831291004670296
9	7632	0.0022821648514260657
10	3089	0.002262828469166777

表 2: block-stripe_result

(二) 调库标准结果

通过调用 NetworkX 库生成库函数计算结果的前 100 个结点排名及 pagerank 值，以下表格只摘取前 10 个结果。

结点排序	结点序号	pagerank 值
1	4037	0.004556412278710509
2	2625	0.003845958019340682
3	6634	0.003721934823415891
4	15	0.0031553234260584245
5	2398	0.00267108237013648
6	2328	0.002617131253537157
7	2470	0.0023842205688539933
8	5412	0.0023820335066259053
9	7632	0.002281121461848156
10	3089	0.0022616729833260453

表 3: standard_result

(三) 结果分析比对

1. TELEPORT=0.85 时运行结果对比

TELEPORT=0.85 时，将输出结果与调库结果进行比对我们发现：基础 PageRank 算法得到的排名结果前 100 名与库函数所得**排名完全相同**，pagerank 值**小数点后 16 位可完全相同**;block stripe 得到的排名结果前 100 名与库函数所得**排名完全相同**，pagerank 值**小数点后 5 位完全相同**。

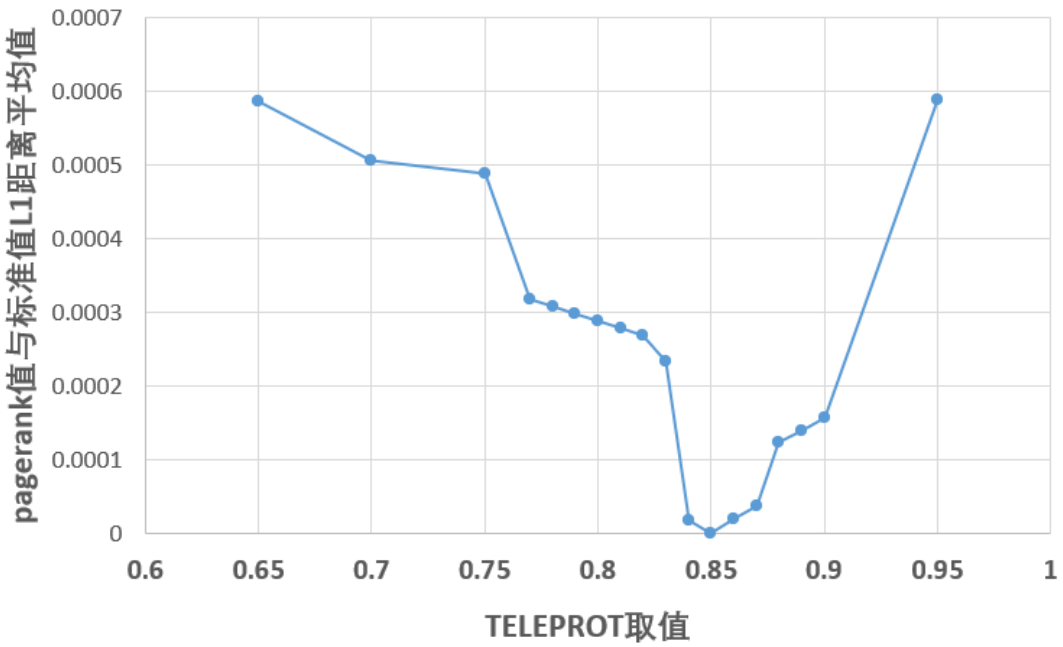
2. 不同 TELEPORT 超参数结果对比

本次对比通过本组编写 compare.py 文件进行对比，输入为两个结果文件，输出为所有结点在两次结果中不同 pagerank 值的 L1 范数距离的平均值，平均值越小代表与调库结果相似度越高，具体实现如下：

```
if __name__ == "__main__":
    # 打开文件，读取数据
    with open('D:\MyEOfNew\bigdata-temp\\basic0.89.txt', 'r') as f:
        lines = f.readlines()

    # 将每一行的两个元素提取出来，放入字典中
    result1 = {}
    for line in lines:
        a, b = line.strip().split()
        result1[a] = b
    # 对字典按照键排序
    sorted_dict1 = {float(k): v for k, v in sorted(result1.items(), key=
        lambda item: float(item[0]))}
```

```
    print(sorted_dict1)

    # 得到排序后的数组v1（其中只包含pagerank值）
    v1 = [sorted_dict1[k] for k in sorted(sorted_dict1.keys())]
    v1 = np.array([float(s) for s in v1])


    with open('D:\MyEOfNew\bigdata-temp\\networkx.txt', 'r') as f:
        lines = f.readlines()
    result2 = {}
    for line in lines:
        a, b = line.strip().split()
        result2[a] = b
    sorted_dict2 = {float(k): v for k, v in sorted(result2.items(), key=
        lambda item: float(item[0]))}
    print(sorted_dict2)

    v2 = [sorted_dict2[k] for k in sorted(sorted_dict2.keys())]
    v2 = np.array([float(s) for s in v2])

    # 计算两次结果中pagerank值的L1范数，并取平均值
    dist = np.linalg.norm(v1 - v2, 1) / 100

    print(dist)
```

实验中选用了 TELEPORT 值从 0.65 到 0.95 的多个数值进行实验，将结果分别与调库得到的标准结果通过 compare.py 文件进行对比，实验结果如下：



TELEPORT 值	pagerank 值与标准值 L1 距离平均值
0.65	0.000585564151409173
0.7	0.0005061287975467829
0.75	0.0004889251537496308
0.77	0.00031696398501974884
0.78	0.0003071658379895508
0.79	0.0002971986579729537
0.8	0.00028723059903282467
0.81	0.0002776679549174438
0.82	0.0002681043263211301
0.83	0.0002324544829550185
0.84	1.8481955598911112e-05
0.85	9.56266316132215e-19
0.86	1.859406416506175e-05
0.87	3.7302385290052815e-05
0.88	0.00012320502804417867
0.89	0.0001397608979820261
0.9	0.0001564172612733427
0.95	0.0005881617740815219

表 4: teleport 参数对比结果表

由以上结果可以得出，当 TELEPORT 为 0.85 时准确率最高，故选用。

五、 实验心得与未来改进

- 本质上来说，Block-stripe 是进行内存优化，但并不会影响实验结果，但在同样的条件下将基础 PageRank 算法优化成 Block-stripe 以后 PageRank 值只能保证小数点后五位和标准调库结果一样，原因未知，有待更深入地探索。
- 在数据集的去重过程中采用 for 循环遍历，复杂度较高，可以通过更加简便的方法进行去重。
- 可以尝试更多的超参数，增加算法的准确度以及分块计算的性能。
- 尝试增加并行计算，使程序在更大规模数据下高效运行。

PageRank 算法的优缺点为：

优点：是一个与查询无关的静态算法，所有网页的 PageRank 值通过离线计算获得；有效减少在线查询时的计算量，极大降低了查询响应时间。

缺点：

- 人们的查询具有主题特征，PageRank 忽略了主题相关性，导致结果的相关性和主题性降低
- 旧的页面等级会比新页面高。因为即使是非常好的新页面也不会有很多上游链接，除非它是某个站点的子站点。

参考文献

[1] Google PageRank. https://www.amsi.org.au/teacher_modules/pdfs/Maths_delivers/Pagerank5.pdf.

[2] PageRank. https://en.wikipedia.org/wiki/PageRank#Internet_use.

[3] PageRank Algorithm-The Mathematics of Google Search. http://pi.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html.