南開大學
Nankai University

计算机学院

大数据计算与应用实验报告

Pagerank 算法的实现与优化

姓名：齐明杰 李佳豪 纪潇洋

专业：信息安全

2023 年 5 月 28 日

1 实验目的

- 实现基础 Pagerank 算法的编写，考虑 dead end 和 spider trap 节点
- 通过优化稀疏矩阵的存储方式提升运算效率
- 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

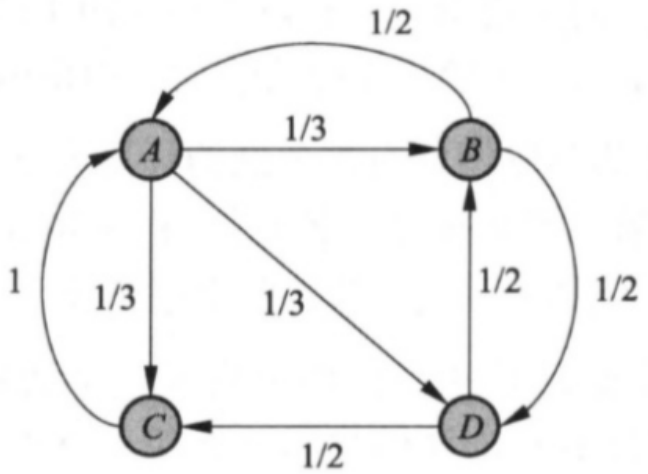
给定输入 N(规模约为 1e4) 个节点间的有向图表示链接关系进行 PageRank 链接分析, 给出所有节点的权重打分, 将结果以 [NodeID] [Score] 的形式存储在 txt 文件中。

3 实验原理

3.1 Pagerank 基本思想

PageRank 是定义在网页集合上的一个函数, 用来计算互联网网页的重要度。其对每个网页给出一个正实数表示网页的重要程度, 整体构成一个向量, PageRank 值越高表明网页越重要, 在互联网搜索的排序中可能就被排在前面。

假设互联网是一个有向图，在其基础上定义随机游走模型，即一阶马尔可夫链，表示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依照连接出去的超链接以等概率跳转到下一个网页，并在网上持续不断进行这样的随机跳转，这个过程形成一阶马尔可夫链。PageRank 表示这个马尔可夫链的平稳分布。每个网页的 PageRank 值就是平稳概率。



上图表示一个有向图，假设是简化的互联网模型，结点 A,B,C 和 D 表示网页，结点之间的有向边表示网页之间的超链接，边上的权值表示网页之间随机跳转的概率。假设有一个浏览者，在网上随机游走。如果浏览者在网页 A，则下一步以 1/3 的概率转移到网页 B,C 和 D。其他节点同理。

3.2 Pagerank 基础模型

给定一个含有 n 个结点的有向图，在有向图上定义随机游走模型，即一阶马尔可夫链，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 n 阶矩阵 M:

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

而转移矩阵具有性质:

$$m_{ij} \geq 0$$

$$\sum_{i=1}^{n} m_{ij} = 1$$

故上图可表示为:

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

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 访问各个结点的概率分布满足:

$$R_{t+1} = MR_t$$

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 R:

$$MR = R$$

$$R = \begin{bmatrix} PR(v_1) \\ PR(v_2) \\ \vdots \\ PR(v_n) \end{bmatrix}$$

R 的各个分量称为各个结点的 Pagerank 值。显然有:

$$PR(v_i) \geqslant 0, \quad i = 1, 2, \cdots, n$$

$$\sum_{i=1}^{n} PR(v_i) = 1$$

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

3.3 Pagerank 算法的优化

3.3.1 dead end 问题和 spider trap 问题

dead end 指的是没有出度的节点，即该节点不会链接到其他的节点。因此这样的节点对网站来说是毫无用处的，因为用户永远无法访问到它们。这些节点可能会影响到 pagerank 的计算结果。

解决 dead end 的方法如下：

1. 给所有页面设置一个初始等级。
2. 把初始等级除以页面的出度总数，分配给它链接到的每个页面。这会把页面之间的贡献相互分摊。
3. 如果一个页面指向了 dead end，则将该页面的等级立即重新分配给其他所有页面，并根据它们中的每一个页面的入度和 pagerank 来重新调整它们的等级。

spider trap 指的是一种特殊的网页结构，即一个爬虫无法从其中一个页面跳转到其他页面，而导致陷入无限循环中。在这种情况下，pagerank 算法会将更多的权值分配给该 spider trap 页面，而忽略其他网页。

为了解决 spider trap 问题，pagerank 算法中通常使用 damping factor（阻尼系数）。damping factor 是一个介于 0 和 1 之间的数字，表示用户浏览网页时停留在当前页面的概率。在 pagerank 算法中，damping factor 用于控制“随机浏览者”不会陷入 spider trap 页面。

具体来说，当 pagerank 算法处理连通图的时候，如果发现某个节点只有出边没有入边，算法会假定该节点能够“达到”其他所有节点，然后对该节点进行 pr 值的迭代。但是，当这个子图是 spider trap 时，这个节点就是一个陷阱。因此，pagerank 算法会使用 damping factor 减少此类情况的影响。

为了解决上述问题，给定一个含有 n 个结点的任意有向图，在有向图上定义一个一般的随机游走模型，即一阶马尔可夫链。一般的随机游走模型的转移矩阵由两部分的线性组合组成，一部分是有向图的基本转移矩阵 M，表示从一个结点到其连出的所有结点的转移概率相等，另一部分是完全随机的转移矩阵，表示从任意一个结点到任意一个结点的转移概率都是 1/n，线性组合系数为阻尼因子 β。这个一般随机游走的马尔可夫链存在平稳分布，记作 R。定义平稳分布向量 R 为这个有向图的一般 PageRank。最终优化后的公式为：

$$r_j = \sum_{i \to j} \beta \frac{r_i}{d_i} + (1 - \beta) \frac{1}{N}$$

一般 PageRank 的定义意味着互联网浏览者，按照以下方法在网上随机游走：任意一个网页上，浏览者或者以概率 β 决定按照超链接随机跳转，这时以等概率从连接出去的超链接跳转到下一个网页；或者以概率 $1 - \beta$ 决定完全随机跳转，这时以等概率 $1/n$ 跳转到任意一个网页。第二个机制保证从没有连接出去的超链接的网页也可以跳转出。这样可以保证平稳分布，即一般 PageRank 的存在，因而一般 PageRank 适用于任何结构的网络。

3.3.2 运算效率问题

- 优化稀疏矩阵的存储: 通过压缩稀疏矩阵，只存储存在非零值的元素和它们的位置信息，可以大大减少所需的存储空间，由于所需空间仅取决于非零条目的数量，因此空间复杂度将接近线性。
- 采用分块的方式 Block-Stripe 提升运算效率。pagerank 的计算涉及大量的矩阵运算，在处理大规模图时会面临较大的计算压力和内存限制问题。block-stripe 方法是一种优化 pagerank 算法的方

法，将邻接矩阵 M 分成小块，每个块仅包含 r^{new} 的相应块中的目标节点，对每个结点进行运算，避免矩阵乘法，可以有效地解决 pagerank 算法处理大规模图时的计算和内存限制问题，优化数据局部性，改善 pagerank 算法的表现。

4 主要工作

1. 实现基础 Pagerank 算法，考虑 dead end 问题和 spider trap 问题
2. 实现稀疏矩阵存储的优化，避免大规模的矩阵运算
3. 实现分块计算和并行处理
4. 实现稀疏矩阵存储和分块计算的联合优化

5 核心代码解析

5.1 基础 Pagerank 算法

5.1.1 核心算法 1: 对数据集的处理

对数据集进行清洗，将文本文件中的有向图转换为以邻接表为存储方式的字典，再统计出该有向图中包含的所有节点。

- 空字典 graph 和一个空集合 allnodes 打开文件 filepath 并按行循环处理文件中的数据，line 代表每一行数据。调用 strip() 方法从 line 中移除换行符并使用 split() 方法将数据拆分为 from node 和 to node 两个整数。
- 将 from node 和 to node 添加到 all nodes 集合中，以保证在矩阵中出现的所有节点都被记录下来。若 [ ] 的键值对添加到字典中，同时将 to node 加入到 from node 对应的值的列表之中；若 from node 已经在字典中，直接将 to node 加入到 from node 对应的值的列表之中。
- 读取完毕后关闭文件，并返回 graph 和 all nodes。其中，graph 是邻接表表示的图，all nodes 是所有节点的集合。

```
import numpy as np
def read_data(file_path):
    graph = {}
    all_nodes = set()
    with open(file_path, 'r') as file:
        for line in file:
            from_node, to_node = map(int, line.strip().split())
            all_nodes.add(from_node)
            all_nodes.add(to_node)
            if from_node not in graph.keys():
                graph[from_node] = []
            graph[from_node].append(to_node)
```

```
    return graph, all_nodes
```

5.1.2 核心算法 2: 利用矩阵乘法实现迭代计算出 Pagerank 值

- 首先构建转移矩阵 s，其中 s[i,j] 表示从 j 到 i 的转移概率。
- 接着对于 s 中每一列进行处理，如果某一列全为 0，则将该列置为 1/n；否则将该列除以其所有元素之和，得到一个新的转移矩阵 m，其中随机跳转参数影响到了矩阵的构造。
- 然后进行迭代更新 pagerank 值，直到满足收敛条件为止。在每次迭代中，通过矩阵乘法计算新的 pagerank 值，并与上一次的 pagerank 值进行比较，判断差异是否达到了收敛阈值。
- 最终得到每个节点的 pagerank 值和每个节点在转移矩阵中的索引。

```
  #由于数据集的结点是不连续的，故重新建立有序结点集
def page_rank_matrix(graph, all_nodes, teleport_parameter, tol):
    """
    基于给定的图、节点和参数，计算PageRank值。
    """
    N = len(all_nodes)
    node_idx = {node: i for i, node in enumerate(sorted(all_nodes))}

    # 构建转移矩阵S
    S = np.zeros([N, N], dtype = np.float64)
    for out_node, in_nodes in 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):
        if (sum_of_col := S[:, col].sum()) == 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

    # 迭代更新PageRank值，直到收敛
    P1 = np.ones(N, dtype = np.float64) / N
    diff, iteration = float('inf'), 1
    while diff > tol:
```

```
        P2 = M @ P1
        diff = np.linalg.norm(P2 - P1)
        print('Iteration: {}, diff: {}'.format(iteration, diff))
        P1 = P2 + (1 - P2.sum()) / N
        iteration += 1

    return P1, node_idx
```

5.2 Sparse Matrix Encoding 算法

- 先根据网络中所有节点数量确定每个节点的初始排名值。
- 按照映射关系遍历每一个节点，若有出度，则遍历每一个出度节点对该节点的贡献值并累加；对于没有出度的节点，直接把旧的 pagerank 值按照比例加到新值上。
- 每次迭代将所有新节点的 pagerank 值归一化（即除以它们的总和），以确保它们仍然表示一个概率分布。
- 在循环中使用停止条件 diff > tol 来控制迭代的停止。其中 diff 是前后两次迭代之间的差异值，而 tol 是设定的容限，指定在迭代过程中算法需要达到的精度，如果 diff 小于 tol 则停止迭代。

5.2.1 稀疏矩阵类的实现

SparseGraph 是我们自定义的稀疏矩阵类，包括初始化稀疏图、访问源节点和目标节点列表、添加从原节点到目标节点的边等多个函数。data 属性是一个以字典形式存储每个源节点及其连接信息的默认字典，all nodes 属性则是用于存储图中的所有节点。

其中，使用了内置模块 defaultdict 来初始化 data 属性。这个字典存储了所有源节点及其对应的出度和目标节点列表。外部可以通过 getitem 方法访问该字典。同时，还提供了多个获取节点信息的方法，如获取源节点出度、目标节点、所有源节点、出度为 0 的节点和所有节点等。而 add edge 则是用来添加从源节点到目标节点的边。

```
class SparseGraph:
    def __init__(self, dtype = np.float64):
        # 初始化稀疏图。
        #参数:
         #   data (dict): 一个表示图的字典，形式是{source_node: (outdegree,
             [destination_nodes])}，默认为 None。
         #   dtype: 数据类型，默认为 np.float64。
        self.data = defaultdict(lambda: (0, []))
        self.all_nodes = set()
        self.dtype = dtype
    def __getitem__(self, source):
        #通过下标访问源节点的出度和目标节点列表。
        #参数: source: 源节点。返回: tuple: 一个元组，包含源节点的出度和目标节点列表。
        return self.data[source]
```

```
    def __contains__(self, source):
        #判断源节点是否在图中。
        #参数: source: 源节点。
        #返回: bool: 如果源节点在图中，返回True，否则返回False。
        return source in self.data
    def __str__(self):
        #将稀疏图转换为字符串表示。
        #返回:str: 稀疏图的字符串表示。
        return f"SparseGraph(data={self.data})"
    def add_edge(self, source, destination):
       # 添加一条从源节点到目标节点的边。参数: source: 源节点。 destination: 目标节点。
        self.all_nodes.add(source)
        self.all_nodes.add(destination)
        if destination not in self.data:
            self.data[destination] = (0, [])
        if source not in self.data:
            self.data[source] = (1, [destination])
        else:
            outdegree, destinations = self.data[source]
            destinations.append(destination)
            self.data[source] = (outdegree + 1, destinations)
    def get_outdegree(self, source):
        #获取源节点的出度.
        #参数: source: 源节点。返回: int: 源节点的出度。
        return self.data[source][0] if source in self.data else 0
    def get_destinations(self, source):
        #获取源节点连接的目标节点列表。
        #参数:source: 源节点。返回: list: 目标节点列表。
        return self.data[source][1] if source in self.data else []
    def get_sources(self):
        #获取所有源节点。返回:list: 源节点列表。
        return list(self.data.keys())
    def get_no_outdegree_nodes(self):
        #获取出度为0的节点。返回:list: 出度为0的节点列表。
        return [node for node in self.data if self.data[node][0] == 0]
    def get_all_nodes(self):
       # 获取所有节点。返回: list: 所有节点的列表。
        return list(self.all_nodes)
```

5.2.2 Pagerank 算法的系数实现

```python
# PageRank 算法的稀疏实现
def page_rank_sparse(G, beta, tol):
    N = len(G.all_nodes)
    ranks = np.array([1 / N] * N, dtype = np.float64)
    node_idx_map = {node: i for i, node in enumerate(G.all_nodes)}
    diff, iteration = float('inf'), 1
    while diff > tol:
        new_ranks = np.array([(1 - beta) / N] * N, dtype = np.float64)
        for node, (out_degree, dests) in G.data.items():
            if out_degree == 0:
                new_ranks += (beta * ranks[node_idx_map[node]]) / N
                continue
            rank_contribution = beta * ranks[node_idx_map[node]] / out_degree
            for dest in dests:
                new_ranks[node_idx_map[dest]] += rank_contribution
        new_ranks /= new_ranks.sum()
        diff = np.sum(np.abs(ranks - new_ranks))
        print(f'Iteration {iteration}: diff = {diff}')
        ranks = new_ranks
        iteration += 1
    return ranks
```

5.3 Block-Stripe 算法

5.3.1 核心算法 1：数据预处理与矩阵分块

- 读取数据文件，找到所有的节点，并计算出每个节点的度；
- 将节点和它们的度按照节点编号升序排列，并将这些节点映射为一个索引，返回 nodes degree 和 node idx；
- 将图中的节点/block 分组，并生成子矩阵；
- 将每个子矩阵保存到一个单独的文件中。

```python
def block_data(blocks=10):
    file_path = '../../Data.txt'
    # 处理数据，得到dead node 并且统计每个节点的度
    nodes = {}
    with open(file_path, 'r') as file:
        for line in file:
            from_node, to_node = map(int, line.strip().split())
            if from_node not in nodes.keys():
                nodes[from_node] = 0
```

```
            if to_node not in nodes.keys():
                nodes[to_node] = 0
            nodes[from_node] += 1
    num_nodes = len(nodes) # 记录节点数量
    nodes_degree = list(sorted(nodes.items(), key=lambda x: x[0])) # 记录度
    node_idx = {node[0]: i for i, node in enumerate(nodes_degree)} # 记录映射

  # 计算每个子图应该包含的节点数
A_block_num = num_nodes // blocks
# 创建空的列表，用于存放所有子图中的边
block_data = [{node: [] for node in nodes} for _ in range(blocks)]
# 打开文件并逐行读取数据，将每条边加入到所属的子图中
with open(file_path, 'r') as f:
    for line in f:
        from_node, to_node = map(int, line.strip().split())
        # 计算出该边属于哪个子图
        block_index = (node_idx[to_node] // A_block_num) if (node_idx[to_node] //
            A_block_num) < (blocks) else blocks - 1
        # 将该边添加到相应的子图中
        block_data[block_index][from_node].append(to_node)
# 如果不存在Block_matrix目录，则创建该目录
if not os.path.exists("./Block_matrix"):
    os.mkdir("./Block_matrix")
# 保存每个子图所包含的所有节点和边到磁盘上
print('--------- 保存矩阵 ----------')
for block in range(blocks):
    save_matrix(block_data[block], block)
# 输出提示信息
print("-------- load all blocks! ---------")
# 返回各个节点的度数、节点索引等相关信息
return nodes_degree, node_idx
```

5.3.2 核心算法 2：读取分块矩阵并实现 Pagerank 的计算

- 初始化参数，包括节点度数、块的数量、传输参数、收敛误差以及迭代次数；
- 在每个块中循环：按行读取该块的稀疏邻接矩阵和一个初始向量 r new；
- 对于每个其他块，计算该块继承 rank 值的总和，并将 delta rank 和所有传递给总矩阵，计算出相应的结果；
- 根据 pagerank 算法的公式，更新每个节点的 pagerank 值，同时考虑 teleport 参数；

- 保存新的 pagerank 向量，并且评估概率向量在两次之间的变化；如果当前的更新幅度低于预设的 tolerance，那么迭代结束。

```
def block_calculate_rank(nodes_degree, node_idx, blocks=10,
    teleport_parameter=0.85, tol=1e-8, iter_num=0):
    num_nodes = len(nodes_degree)
    while True:
        theta = 0
        iter_num += 1
        for block in range(blocks):
            # 读取 Block Matrix 初始r_new
            graph = read_matrix(block)
            r_new = np.array(read_vector(block, True))
            block_base = num_nodes // blocks * block

            for p_block in range(blocks):
                base = num_nodes // blocks * p_block # 计算全局下标
                r_old = np.array(read_vector(p_block))

                for rx, weight_rx in enumerate(r_old):
                    rx_global_idx = rx + base # 全局下标
                    rx_name = str(nodes_degree[rx_global_idx][0])
                    # 模拟取Matrix中的一条
                    degree_rx = nodes_degree[rx_global_idx][1]
                    if degree_rx > 0:
                        for destination in graph[rx_name]:
                            des_idx = node_idx[destination] - block_base
                            r_new[des_idx] = (teleport_parameter * weight_rx) /
                                degree_rx + r_new[des_idx]

                    else:
                        # dead node
                        r_new += teleport_parameter / num_nodes * weight_rx

            r_old = np.array(read_vector(block))
            save_vector(r_new.tolist(), block, False)

            # 判断收敛
            theta += np.abs(r_new - r_old).sum()
        if theta < tol:
            print(f'迭代{iter_num}次')
            break
```

5.4 逆矩阵求解算法

代数算法可以通过一般转移矩阵的逆矩阵计算求有向图的一般 PageRank。

由：

$$(I - dM)R = \frac{1-d}{n}1$$

$$R = (I - dM)^{-1}\frac{1-d}{n}1$$

这里 I 是单位矩阵。当 0<d<1 时，上述线性方程组的解存在且唯一。这样，可以通过求逆矩阵 $(I - dM)^{-1}$ 得到有向图的一般 PageRank。

- 获取所有节点的数量 n，并将所有节点按字母表排序。建立节点和索引之间的映射 node idx。
- 根据输入的图构建转移矩阵 s。对于每一项 (graph.items()) 中的出度节点 (out node) 和与其相连的入度节点 (in nodes), 将对应位置的元素设为 1，即 s[node idx[in node], node idx[out node]] = 1.
- 处理矩阵。如果当前列的总和为 0，就将这一列中的所有元素都设为 1/n（平均分配权重）；否则将这一列中的所有元素都除以总和。处理后的矩阵为 s，表示从一个页面跳到另一个页面的概率。
- 初始化单位矩阵 e 和一个全为 1 的列向量 et，用来创建随机浏览模型 d。直接使用逆矩阵方法计算结果 p。计算公式为 p = 随机浏览模型 * a + s * (1-a)，其中 a 为迭代系数，随机浏览模型 d 等价于 ((1 - teleport parameter) / n * et) 。

```python
def page_rank_reverse(graph, all_nodes, teleport_parameter = 0.85):
    """
    基于给定的图、节点和参数，使用逆矩阵方法计算PageRank值。
    """
    N = len(all_nodes)
    node_idx = {node: i for i, node in enumerate(sorted(all_nodes))}

    # 构建转移矩阵S
    S = np.zeros([N, N], dtype = np.float64)
    for out_node, in_nodes in 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):
        if (sum_of_col := S[:, col].sum()) == 0:
            S[:, col] = 1 / N
        else:
            S[:, col] /= sum_of_col

    E = np.identity(N, dtype = np.float64)
```

```
22      ET = np.ones((N, 1), dtype = np.float64)
23      # 直接使用逆矩阵计算结果
24      P = np.linalg.inv(E - teleport_parameter * S) @ ((1 - teleport_parameter) / N *
            ET).flatten()
25
26      return P, node_idx
```

6 实验结果

我们用自己编写的代码对数据集中各个节点的 Pagerank 进行计算，我们发现在不同算法中，Pagerank 值的前六名保持不变，而所有点的 Pagerank 值或多或少出现偏差。由于篇幅限制，我们在这里只给出 Top10 的计算结果，完整结果在附件中呈现。

表 1-4 分别是基础 Pagerank 算法、Sparse Matrix Encoding 算法、Block-Stripe 算法、逆矩阵法的计算结果，表 5 是通过调用 networks 包得到的结果，作为正确率的参考，实验结果表明，以我们的 networks 的计算结果作为标准时，前 100 名的节点均大致相同，说明我们采取的算法正确率较高。而最终的准确率由老师与真正的标答进行核对，这里不予给出。

在实验中，我们所实现的矩阵分块 block-stripe 算法能够实现分块处理数据，在数据量极大的情况下能够节约内存空间，采用多块并行计算时能够降低运行时间。这是由于：

1. 局部性原理：block-stripe 算法为每个块（或一组相邻的块）单独计算 pagerank 值，而不是整个矩阵上的操作，则之间至少要保证相邻块所需要的数据要尽量同时存在于 cache 中，避免反复从主存到 cache 的调取。

2. 多线程并行计算：stripe-block 算法可以同时利用多个处理器核心进行计算，极大地加速了计算的过程。算法实现方法会考虑如何切割问题、并行调用矩阵计算，并合并结果，降低了计算时间和系统负载。

而在本次实验所给的数据集中，我们无法显著体现出分块计算的优势，但成功实现的分块算法能够在处理规模较大的数据时发挥巨大作用。

6.1 基础 Pagerank 算法计算结果

表 1: 基础 Pagerank 算法计算结果

排名	结点序号	Pangerank 值
1	4037	0.004550721327537371
2	2625	0.0038388957853952143
3	6634	0.0037939505222215446
4	15	0.0031500476814941706
5	2398	0.0026700013520133924
6	2328	0.002612516730784656
7	5412	0.002380151035317216
8	2470	0.0023784726492110503
9	7632	0.002280095616409648
10	3089	0.002257379115001626

6.2 Sparse Matrix Encoding 算法计算结果

表 2: Sparse Matrix Encoding 算法计算结果

排名	结点序号	Pangerank 值
1	4037	0.004989267501122889
2	2625	0.004070535049315213
3	6634	0.003726160655316386
4	15	0.0030983201045671374
5	2398	0.002814926970103211
6	2328	0.0025752685475940542
7	2470	0.002543549706483492
8	3089	0.0024706916971194252
9	6946	0.0023736645355066505
10	3352	0.002363092056879781

6.3 Block-Stripe 算法计算结果

表 3: Block-Stripe 算法计算结果

排名	结点序号	Pangerank 值
1	4037	0.00498926754769378
2	2625	0.004070535095354542
3	6634	0.0037261606943220084
4	15	0.0030983201398049765
5	2398	0.0028149270035836547
6	2328	0.002575268575674095
7	2470	0.0025435497290086364
8	3089	0.0024706917246012194
9	6946	0.002373664563943835
10	3352	0.0023630920829801638

6.4 逆矩阵法计算结果

表 4: 逆矩阵法计算结果

排名	结点序号	Pangerank 值
1	4037	0.004550721327537371
2	2625	0.0038388957853952143
3	6634	0.0037939505222215446
4	15	0.0031500476814941706
5	2398	0.0026700013520133924
6	2328	0.002612516730784656
7	5412	0.002380151035317216
8	2470	0.0023784726492110503
9	7632	0.002280095616409648
10	3089	0.002257379115001626

6.5 networks 求解结果（参考）

表 5: networks 求解结果

排名	结点序号	Pangerank 值
1	4037	0.0045507214010943615
2	2625	0.0038388958552070633
3	6634	0.0037939506035290977
4	15	0.0031500477379419974
5	2398	0.002670001403031027
6	2328	0.0026125167773243716
7	5412	0.0023801510834965587
8	2470	0.0023784726838275284
9	7632	0.0022800956608540167
10	3089	0.0022573791579518277

致 谢

感谢杨老师耐心细致的讲解，您深入浅出的分析让我们对 Pagerank 算法有了清晰的认知并对大数据这个领域产生了浓厚的兴趣。

感谢助教李学长的耐心协调和答疑，为我们排忧解难，更加严谨地从事学术活动。

谨以此报告代表我对大数据计算与应用这门课的喜爱以及对学问的敬重！