PageRank作业报告

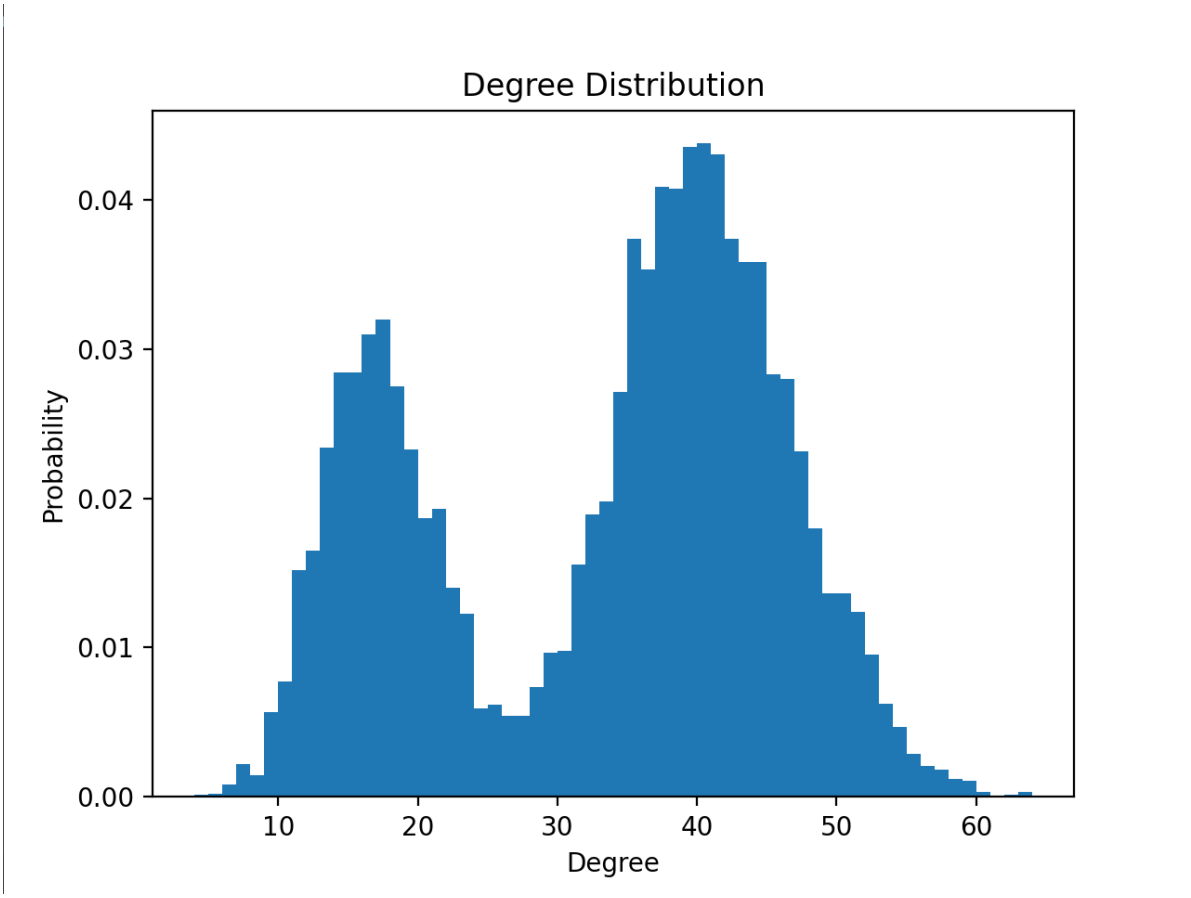
2011273 姚翔载 2011440 杨馨仪

数据集说明

数据集为给定的数据集，包含一个txt格式的文件。其中每行包括两个int型数字，用空格隔开。它们分别表示当前节点Id和指向节点Id。

对数据集进行统计，可以发现如下数据：

- 节点数量: 8297
- 边的数量: 135737
- 聚类系数: 0.004983081304261809
- 度的分布如下：



可以发现虽然有8297个节点，但每个节点最多的度约60个左右，且多为18个和42个左右，因此可以断定邻接矩阵是非常稀疏的。

任务完成情况

1. 完成基本PageRank算法，包括PageRank矩阵迭代计算流程、程序收敛判断；
2. 针对Dead Ends节点，将其等概率链接到其他所有节点；针对Spider Trap节点，使用random teleport解决；
3. 实现block-stripe update，包括理论分析以及M和r的block的构建

4. 使用Python完成代码，包括读取、运算和输出；完成实验报告的撰写

关键代码细节与原理

1. 数据读取：根据txt文件的特性对数据进行读取。考虑到需要对稀疏矩阵进行优化，这里采用类似邻接链表的形式读取数据并存储。数据结构如下：

```
class nodeGraph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, from_node, to_node):
        if from_node not in self.graph:
            self.graph[from_node] = []
        self.graph[from_node].append(to_node)

    def print_graph(self):
        for node in self.graph:
            print(node, '->', self.graph[node])
```

   其中每个nodeGraph对象都有一个字典，其中每一项的query为一个int型数字，代表节点Id；每一项的key为这个节点指向的其他边。通过使用邻接矩阵的形式进行读取和存储能够有效对稀疏矩阵进行优化，节省存储空间。

2. 构建链接矩阵，首先展示直接构建完整的链接矩阵到内存中：

```
def construct_link_matrix(graph, num_nodes, tp=0.85):
    link_matrix = np.zeros((num_nodes, num_nodes))

    for from_node, to_nodes in graph.graph.items():
        for to_node in to_nodes:
            link_matrix[to_node][from_node] = 1

    # 处理"Dead End"节点，将其等概率链接到所有其他节点，并乘以阻尼因子
    dead_end_nodes = [node for node in range(0, num_nodes) if node not in
graph.graph]
    for node in dead_end_nodes:
        link_matrix[:, node] = 1

    # 归一化
    link_matrix = normalize_columns(link_matrix)
    return link_matrix
```

   首先构建nxn的矩阵，根据graph数据结构的字典，找到每个节点及其指向的节点，将有向图的邻接矩阵的对应节点处标为1。考虑到Dead End节点，即没有任何出链的节点，一种处理方法是将其等概率链接到其他节点。因此，直接找不在数据结构中的节点，然后将一列赋值为等概率即可（最后进行归一化）。考虑到Spider Trap节点，一种方法是使用teleport parameter使每个节点都有概率随即跳转，而不是一定跟随链接节点。为每一个点赋值一个teleport parameter并归一化即可。

   为了节省存储空间（并为后续条块化做准备），也可以直接通过邻接链表构建部分邻接矩阵，只需要选取想构建的范围即可。一种实现如下：

```
def construct_part_link_matrix(graph, num_nodes, ina, inb, tp=0.85):
    link_matrix = np.zeros((inb-ina, num_nodes))

    for from_node, to_nodes in graph.graph.items():
        for to_node in to_nodes:
            if to_node in range(ina, inb):
                link_matrix[to_node-ina][from_node] = 1

    # ... 其余操作
    return link_matrix
```

可以看到这样能直接选取ina到inb的输入节点的邻接矩阵。

3. 基础PageRank

```
def compute_pagerank(link_matrix, epsilon=1e-8, max_iterations=1000):
    # naive pagerank + spider trap/dead ends handling
    num_nodes = link_matrix.shape[0]
    pagerank = np.ones(num_nodes) / num_nodes

    for _ in range(max_iterations):
        new_pagerank = (1 - tp) / num_nodes + \
                               tp * np.dot(link_matrix, pagerank)
        if np.linalg.norm(new_pagerank - pagerank) < epsilon:
            return new_pagerank
        pagerank = new_pagerank

    return pagerank
```

在存储空间允许的情况下，通过在内存加载所有邻接矩阵就可以实现pagerank。首先输入链接矩阵，并初始化一个等概率的PageRank向量表示初始状态，其长度为节点数量。通过矩阵乘法，每个节点都通过随机等概率地在指向的节点中选择，从而在图上游走、达到下一个状态。为了处理Spider Trap节点，设置每个节点都有一定概率即1-teleport number跳转到任意节点上。迭代上述过程直到收敛即可。收敛的判断条件为更新前后向量的欧氏距离小于一个阈值，我们设定为1e-8。

4. Block-stripe更新的pagerank

原理：根据本科课件及Stanford PageRank课件，Block-stripe更新主要有两个concern：

1. r_new不能很好地适配内存，因此要将r_new分块；
2. M往往比r大得多，因此也应分块

具体来说，可以按照一个block_size将M和r_new尽量分成大小相同的blocks。为了保证PageRank的合理运算和矩阵的正确运算，M和r的block size应相同且一一对应。根据PageRank的原理，PageRank可以看作一个马尔可夫链，下一刻的PageRank分数可以看成当前节点按概率随机游走的结果。由此，要通过矩阵运算计算block-wise的下一刻状态，假设我们将block-wise的节点的index范围表示为i到j，当前节点表示为bn，那么bn的PageRank的更新可以简单表示为：

$$\sum(\text{节点}i\text{的}PageRank\text{值} \cdot \text{节点}i\text{访问}bn\text{的概率})$$

根据矩阵乘法的原理，要想对bi到bj这个block中的所有节点进行更新，可以简单地用矩阵乘法表示为：

$$r^{new}[i:j] = M[i:j] \cdot r^{old}$$

其中"·"表示点乘。这样只需要M的i到j存储的是由i到j的block_size个节点作为destination node的记录即可，即第i行的m项表示m到i的链入情况。

这样，我们就可以将M分成存储destination nodes的stripe，即“Each stripe contains only destination nodes in the corresponding block of r^new”（from：Stanford课件）。这些stripes可以被存储到分布式系统中并且可以在分布式系统中并行读取，不需要全部同时加载到同一个内存中，大大增大了计算效率并减少单个计算节点的内存负载，在应对大规模数据时尤其有效。然而这种方式的缺点也是显著的，包括频繁的通信可能增大负载等

对于每个stripe的计算，将其与r^old点乘即可。最后（可能需要通信后）拼接所有r^new。

这部分的代码为：

```
def block_stripe_pagerank(M_list, r_list, epsilon=1e-8,
max_iterations=1000):
    num_nodes = M_list[0].shape[1]
    r_old =  np.concatenate(r_list, axis=0)

    # Block-Stripe更新算法
    for _ in range(max_iterations):
        new_pagerank = np.zeros(num_nodes)
        block_idx = 0
        for M_cur in M_list:
            r_cur = (1 - tp) / num_nodes + \
                            tp * np.dot(M_cur, r_old)
            r_list[block_idx] = r_cur
            block_idx += 1
        r_new = np.concatenate(r_list, axis=0)
        if np.linalg.norm(r_new - r_old) < epsilon:
            return r_new
        r_old = r_new

    return pagerank
```

同样使用random teleport来解决spider trap的问题。M_list与r_list分别为存储M的stripe和r的block的python列表。在这部分的代码中我们将其放在内存中，通过读取列表对应项来模拟分布式系统中分别读取。也可以将这部分换为从磁盘中读取的接口。总之，不论是从内存还是磁盘中读取，分块计算的步骤是相同的。

对于M_list的构建，我们给出两种实现：

第一种是直接从完整的M中切片。虽然这种方法实际上还是将整个M放在同一个内存中来处理，但在内存允许的情况下，避免了不必要的频繁访问M stripe磁盘存储或频繁构建M stripe的问题，计算效率较高

```
def simulate_store_link_matrix(link_matrix,num_nodes,block_size):
    M_list = []
    for block_start in range(0, num_nodes, block_size):
        block_end = min(block_start + block_size, num_nodes)
        # 如有需要，可以将block保存到磁盘
        # 由于要求可以在内存里对block运算，为了代码实现的简洁，这里
        # 用一个list存储在内存中
        M_list.append(link_matrix[block_start:block_end])
    return M_list
```

或者我们也可以采用**第二种**方法，即每次都从磁盘中读取或每次都从高效率的存储结构中构建stripe，来更好地模拟分布式系统中block-stripe计算PageRank的真实情况。这里为了更好地体现对稀疏矩阵的优化，我们使用邻接链表直接构建stripe：

```
def block_stripe_pagerank_read(num_nodes, block_size, r_list, tp=0.85,
epsilon=1e-8, max_iterations=1000):
    r_old =  np.concatenate(r_list, axis=0)

    # Block-Stripe更新算法
    for _ in range(max_iterations):
        new_pagerank = np.zeros(num_nodes)
        block_idx = 0
        for block_start in range(0, num_nodes, block_size):
            block_end = min(block_start + block_size, num_nodes)
            M_cur = construct_part_link_matrix(graph, num_nodes,
block_start, block_end,tp)
            r_cur = (1 - tp) / num_nodes + \
                                tp * np.dot(M_cur, r_old)
            r_list[block_idx] = r_cur
            block_idx += 1
        r_new = np.concatenate(r_list, axis=0)
        if np.linalg.norm(r_new - r_old) < epsilon:
            return r_new
        r_old = r_new

    return pagerank
```

construct_part_link_matrix即为根据block的idx构建stripe，实现也较简单，根据邻接链表，找到destination在为block范围内的节点的index并记录即可：

```
def construct_part_link_matrix(graph, num_nodes, ina, inb):
    link_matrix = np.zeros((inb-ina, num_nodes))

    for from_node, to_nodes in graph.graph.items():
        for to_node in to_nodes:
            if to_node in range(ina, inb):
                link_matrix[to_node-ina][from_node] = 1

    # 处理"Dead End"节点，将其等概率链接到所有其他节点
    dead_end_nodes = [node for node in range(0, num_nodes) if node not in
graph.graph]
    for node in dead_end_nodes:
        link_matrix[:, node] = 1
```

```
    # 归一化
    link_matrix = normalize_columns(link_matrix)
    return link_matrix
```

对于我们最终展示的实验结果，由于每次都从邻接链表来构建会非常耗时，为了在短时间内印证分块运算和邻接链表构建矩阵的正确性，我们最终采用如下代码进行测试，其中邻接链表构建矩阵只构建一次：

```
def simulate_store_link_matrix(graph,num_nodes,block_size):
    M_list = []
    for block_start in range(0, num_nodes, block_size):
        block_end = min(block_start + block_size, num_nodes)
        M_list.append(construct_part_link_matrix(graph, num_nodes, block_start,
block_end))
    return M_list

def block_stripe_pagerank(M_list, r_list, tp=0.85, epsilon=1e-6,
max_iterations=1000):
    num_nodes = M_list[0].shape[1]
    r_old =  np.concatenate(r_list, axis=0)

    # Block-Stripe更新算法
    for _ in range(max_iterations):
        new_pagerank = np.zeros(num_nodes)
        block_idx = 0
        for M_cur in M_list:
            r_cur = (1 - tp) / num_nodes + \
                                tp * np.dot(M_cur, r_old)
            r_list[block_idx] = r_cur
            block_idx += 1
        r_new = np.concatenate(r_list, axis=0)
        if np.linalg.norm(r_new - r_old) < epsilon:
            return r_new
        r_old = r_new

    return pagerank
```

实验结果及分析

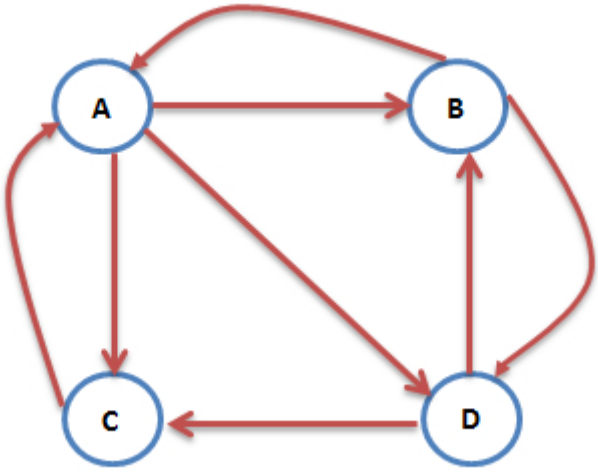
实验设置：

teleport number为0.85，判断收敛的epsilon=1e-8，block数设为100

结果分析：

1. 在Pagerank_v_100和Pagerank_bs_200分别记录普通的和使用block-stripe更新的结果。两种方式的结果相同，证明了矩阵运算的正确性。
2. PageRank运算结果的正确性：

   考虑一个简单的样例：



经过计算，该PageRank会收敛到如下值 （图片均来自cnblogs）：

$$V_1 = MV_0 = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 15/48 \\ 11/48 \\ 11/48 \\ 11/48 \end{bmatrix} \cdots \begin{bmatrix} 11/32 \\ 7/32 \\ 7/32 \\ 7/32 \end{bmatrix} \cdots \begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

上述结果是不考虑random teleport的收敛值，因此将我们代码中的tp设为1，随后输出结果：

Rank 1: Node 0 - PageRank Score: 0.3333333358168602
Rank 2: Node 3 - PageRank Score: 0.22222222139437992
Rank 3: Node 2 - PageRank Score: 0.22222222139437992
Rank 4: Node 1 - PageRank Score: 0.22222222139437992

可以说明基础PageRank算法流程的正确性。

将结果与networkx的输出结果进行对比：

我们的top100结果为[2730, 7102, 1010, 368, 1907, 7453, 4583, 7420, 1847, 5369, 3164, 7446, 3947, 2794, 3215, 5346, 7223, 630, 4417, 4955, 3208, 2902, 5671, 5833, 5553, 8096, 3204, 758, 6301, 5769, 8194, 4957, 8060, 7938, 5584, 6568, 1430, 7250, 3185, 2737, 3751, 150, 5099, 2944, 7872, 2639, 5074, 1034, 229, 6648, 4222, 7406, 2464, 3578, 930, 6777, 2484, 4944, 1197, 3221, 2041, 7579, 6787, 6530, 8112, 6005, 6190, 5655, 251, 3951, 8018, 233, 2589, 5996, 482, 972, 7499, 7442, 1173, 2369, 6315, 5129, 7784, 5998, 4692, 4255, 6692, 4832, 5275, 5376, 2232, 6928, 260, 1677, 6847, 6883, 7702, 1798, 4681, 2664]

networkx的结果为[2730, 7102, 1010, 368, 1907, 7453, 4583, 1847, 7420, 5369, 3164, 7446, 3947, 2794, 3215, 5346, 7223, 630, 4417, 4955, 3208, 2902, 5671, 5833, 5553, 8096, 3204, 758, 6301, 5769, 8194, 4957, 8060, 7938, 5584, 1430, 6568, 7250, 3185, 2737, 3751, 150, 5099, 2944, 7872, 5074, 2639, 1034, 229, 6648, 4222, 7406, 3578, 2464, 930, 6777, 2484, 4944, 1197, 2041, 3221, 7579, 6787, 8112, 6530, 6005, 6190, 5655, 251, 8018, 3951, 233, 2589, 5996, 482, 972, 7442, 7499, 1173, 2369, 6315, 5998, 5129, 4692, 7784, 4255, 6692, 4832, 5275, 5376, 2232, 6928, 260, 1677, 6847, 7702, 6883, 1798, 4681, 2664]

可以发现只有少数的节点判断错误，考虑到networkx包具有额外的实现，且由于处理spider trap等方式的不同（例如有些实现直接使用random teleport同时处理dead end和spider trap），这种误差是可以接受的

3. 运行时间比较，对于每种算法，测量其从读取数据到完成PageRank运算完成的时间：

   networkx包：0.7370s

   普通PageRank（具有Dead end和Spider trap处理）：2.1224s

   block-stripe PageRank：35.70460748672485s

   可以发现普通的PageRank实现仍有很大的运算优化空间，包括使用更高效的数据结构、更高效的分块矩阵运算、并行优化等。

4. 由于我们的实现中，并未将M和r放在磁盘中读取而是直接在内存上运算，因此不直接测量程序运行占用的内存，而是通过计算一些数据结构的内存占用量来进行分析：通过sys.getsizeof对一些数据结构占用给定内存进行分析：

   graph的字典，即邻接矩阵：288KB

   PageRank向量，使用numpy array：64KB

   完整的链接矩阵：525MB

   由此可见，对稀疏矩阵进行优化例如使用邻接链表等数据结构存储等方法对于节省内存、降低负载是非常有必要的；对于PageRank算法来说，使用条块更新法也能在牺牲一些通信代价的基础上，将大规模数据成功地部署到分布式系统上进行运算

5. 不同tp值的影响，如下表所示。tp值即teleport值，它越小，收敛所需时间越少，但pagerank值也随之减少。这是由于图中部分spider node造成的。例如，完全不加抑制的话，迭代时在一些spider node中循环，则需要更长的时间以达到稳定状态甚至无法正常收敛；如果将tp设置得越小，表明每次浏览后等概率跳到随机节点的概率越大、按照图结构浏览的概率越小，最后更倾向于均匀分布，因此pagerank的值和方差会随之降低。此外，根据与networkx输出的对比，发现tp值过大或过小，准确率都会降低。这是因为，过大时无法很好地处理spider traps，过小时偏向均匀分布的错误结果。因此，选择一个合适的tp值至关重要。

tp	time	top_score	mean	var
1	6.1086	3.05E-03	2.49E-03	3.60E-08
0.85	2.1224	8.72E-04	7.26E-04	2.31E-09
0.6	1.2544	3.67E-04	3.18E-04	2.38E-10
0.4	0.9243	2.36E-04	2.12E-04	5.07E-11
0.1	0.7229	1.35E-04	1.36E-04	1.46E-12