南開大學
Nankai University

计算机学院

大数据计算及应用

PageRank 实验报告

姓名：王娇妹

学号：2012679

专业：计算机科学与技术

2023 年 5 月 16 日

目录

1 实验内容

基于给定数据集，计算出得分排名前 100 的节点，可采用不同 teleport 参数，必须给出 teleport 参数为 0.85 时的结果。除了实现基本的 PageRank 算法，还需实现 Block-Stripe Update 算法。

实验要求：

1. 语言: C/C++/JAVA/Python
2. 考虑 dead ends 和 spider trap 节点
3. 优化稀疏矩阵
4. 实现分块计算
5. 程序需要迭代至收敛
6. 不可直接调接口，例如实现 pagerank 时，调用 Python 的 networkx 包
7. 结果格式 (.txt 文件)：[NodeID] [Score]

2 数据集说明

该数据集中包含了 8297 个节点之间的 83852 个链接关系。

每行数据表示一个链接关系，以两个节点的 ID 标识，其格式为：FromNodeID ToNodeID，表示由 FromNodeID 链向 ToNodeID。

```
83847   2932 3724
83848   228 415
83849   2660 3464
83850   5543 6953
83851   280 1014
83852   1847 2398
```

图 2.1: Data.txt 中链接个数

3 实验原理

对于每个网页 i ，计算其 PageRank 值，可以采用以下公式进行计算：

$$PR(i) = \frac{(1-d)}{N} + d \cdot \sum_{j \in M_i} \frac{PR(j)}{L(j)}$$

其中，$PR(i)$ 表示网页 i 的 PageRank 值，N 表示网页总数，d 为阻尼系数，M_i 表示所有指向网页 i 的网页集合，$PR(j)$ 表示网页 j 的 PageRank 值，$L(j)$ 表示网页 j 的出链数。

4 basic PageRank 算法

4.1 数据读取

在数据读取步骤，需要将 txt 文档中的数据读到内存中，并统计节点的数量。

在该实验数据集（Data.txt）中，每行数据的格式为：FromNodeID ToNodeID。每行代表一条边，以两个节点的 ID 标识，表示由 FromNodeID 链向 ToNodeID。

首先创建一个变量 node_num，初始化为 0，用于记录节点总数。

使用 readlines() 方法从文件对象中读取所有行，接下来遍历每一行，使用 split() 方法将字符串分割成子字符串，得到 FromNodeID 和 ToNodeID ，然后其转化成整数类型，存储到稀疏矩阵中。与此同时，记录当前的最大节点索引为 max(node_num, FromNodeID, ToNodeID) ，循环结束，可以获得节点的总数。

4.2 稀疏矩阵

在 basic PageRank 算法中，内存中需要存储所有的节点及其链接关系，如果直接存储转移矩阵 M ，由于 M 非常稀疏，不仅耗费资源较多，效率也会很低。

由于 M 存储所需的空间与非零元素的数目呈线性关系，所以使用稀疏矩阵来存储 M，对 M 进行如下压缩处理：

1. 只存储有出链的节点。
2. 使用字典来存储节点和链接关系。key 为链向其他节点的节点 ID，value 为列表格式，存储该节点链向的所有节点 ID。

数据读取和稀疏矩阵关键代码

```
    with open(data_path) as f:
        lines = f.readlines()
        for line in lines:
            # get id int
            id_2_str = line.split()
            l_node_idx = int(id_2_str[0])
            r_node_idx = int(id_2_str[1])
            # 节点总数
            node_num = max(node_num, l_node_idx, r_node_idx)
            #print("now the max node idx is ", node_num)
            # 节点链向的 ID 加入列表
            if l_node_idx not in n2n_dic.keys():
                n2n_dic[l_node_idx] = [r_node_idx]
            else:
                n2n_dic[l_node_idx].append(r_node_idx)
```

4.3 Dead ends 和 Spider trap 节点

Dead ends 是指某个节点没有外向链接，也就是说该节点无法到达其他节点；Spider trap 是指一个节点链接了很多其他节点，但是没有外向链接，这些节点只能被访问到，而无法离开。这两种节点会导致计算出来的 PageRank 值出现偏差。

本实验中，我采用随机跳转的方式进行处理。如果当前节点没有外向链接，那么用户会以一定的概率跳转到其他任意节点，从而使得 Dead ends、Spider trap 不再是 “有进无出”，避免 PageRank 值的偏差。

在每次迭代更新节点的 score 列表时，每个页面的 PageRank 值会被乘以阻尼因子 d，同时会加上一个随机跳转概率 (1-d)/N ，其中 N 是页面总数。

4.4 计算 score

4.4.1 初始化 score

在给定数据集中，每个节点以数字标识，从 1 开始。

我使用列表来存储每个节点的 score。为便于查看和使用，将 score 列表长度设置为 node_num + 1，不使用索引为 0 的元素，并将其设为 0，其余元素初始化为 1/node_num。

ID 为 k 的节点的得分为 score[k]。(k = 1，2，3...)

4.4.2 更新 score

1. 进入条件为 True 的 while 循环，记录当前轮数 i。
2. 将当前 score 列表复制一份，作为上一轮的得分，用于计算新旧两轮得分之差的绝对值之和，作为判断停止条件的指标。
3. 循环遍历数据读取阶段的稀疏矩阵，该稀疏矩阵使用字典的形式保存，key 为出链节点，value 为链向的所有节点的列表。
4. 对于该字典的每个键值对，通过 len() 函数计算 value 列表长度，记为该节点的出链节点总数。然后遍历 value 列表，为该列表中每个节点的 score 加上 score[key]/lenth 。其中 key 为出链节点，lenth 为该出链节点所链向的节点总数。
5. 归一化。计算 score 列表的和 sum_score，将 score 中的每个元素除以 sum_score。(score[0] 不处理，因为它不表示节点)
6. 加入阻尼因子和随机跳转概率，得到新的 score 列表，这是新一轮得分列表。
7. 计算步骤 2 中的上一轮得分和当前轮得分之差的绝对值之和 dif_sum。
8. 判断停止条件：如果达到设定的最大循环轮数或者 dif_sum 小于阈值，循环停止。如果不满足，回到步骤 2 继续执行。

update score 关键代码

```
    # 初始化 不用idx为0的元素, node num+1
    score_ini = [1/float(node_num)] * (node_num + 1)
    score_ini[0] = 0
    # last round score
    last_score = score_ini.copy()
    # 最多计算 count 轮
    i = 0
    while True:
        for from_node, to_list in n2n_dic.items():
            to_num = len(to_list)
            for to_node in to_list:
                score_ini[to_node] += last_score[from_node]/float(to_num)
        # normalize
        sum_score = sum(score_ini)
        for j in range(1, node_num+1):
```

```
            score_ini[j] = score_ini[j]/sum_score
        # add alpha
        dif_sum = 0.0
        for j in range(0, node_num+1):
            if j == 0:
                continue
            score_ini[j] = score_ini[j] * K + (1 - K)/float(node_num)
            dif_sum += abs(score_ini[j] - last_score[j])
        print("dif_sum =", dif_sum)
        i = i + 1
        if dif_sum <= convergence_threshold or i == count:
            print("stop")
            print(sum(score_ini))
            break
        else:
            last_score = score_ini
```

4.5 输出 score

在“更新 score”步骤，得到了所有节点的 score 列表，但该列表是以节点 ID 排序的，还需要对每个节点的 score 自高到低排序，获取得分最高的 100 个节点的 ID 和得分，每个节点数据以 [NodeID] [Score] 的格式作为一行，保存到本地 txt 文件。

4.6 重要参数

以下是本实验中涉及几个重要参数。

1. 阻尼因子 d，设置为 0.85。
2. 收敛阈值。当 PageRank 值的变化不再超过这个给定的阈值时，认为 score 列表收敛，并将其作为最终结果。此实验中，设置收敛阈值为 0.00001。
3. 最大循环次数，表示在更新 score 列表时的最多循环多少轮的计算，设置为 50。在多次实验中，该 Basic PageRank 算法基本都在 20 轮循环以内达到收敛状态。

4.7 结果分析

4.7.1 Basic PageRank 算法与调用库函数对比

下面是阻尼因子 d = 0.85，收敛阈值为 0.00001 时的得分最高的前 20 个节点。

```
4037   0.004952806901050517
2625   0.0034516500839220335
6634   0.0031850468920015577
2470   0.00298834175945143
15     0.0026501829657117016
2328   0.0021524952373620414
2398   0.0021481965927422153
6774   0.0020262707449358315
```

```
2237   0.0019752308854415596
1186   0.0019729033476980876
665    0.0019547086659999283
3352   0.0019208342145150183
4191   0.001862590359359267
7553   0.0018594744052468858
5254   0.0018381923861955057
3089   0.0017887417965847838
6832   0.0016829078541022884
4875   0.0016792878781378836
5412   0.0016378704307428045
4310   0.0016286827057879069
```

下面是调用 networkx 包，设置阻尼因子为 0.85，收敛阈值为 0.00001 的得分最高的前 20 个节点。

```
4037 0.004539372313144959
2625 0.003822752995504555
6634 0.0035024970697229356
15 0.0031437648620196816
2398 0.0026484245422519854
2328 0.002602618599956839
2470 0.002397651464818284
5412 0.0023584684412651406
3089 0.002281899223203962
7632 0.002273191393070048
3352 0.0022067291280423577
737 0.002185048440574197
4191 0.0021389444464734305
3456 0.002138036073200611
2237 0.0021374008565678533
7553 0.0021099159157061246
5254 0.002104609372935957
6832 0.0020853357491931465
2066 0.0020221828437291807
1297 0.001991260866821734
```

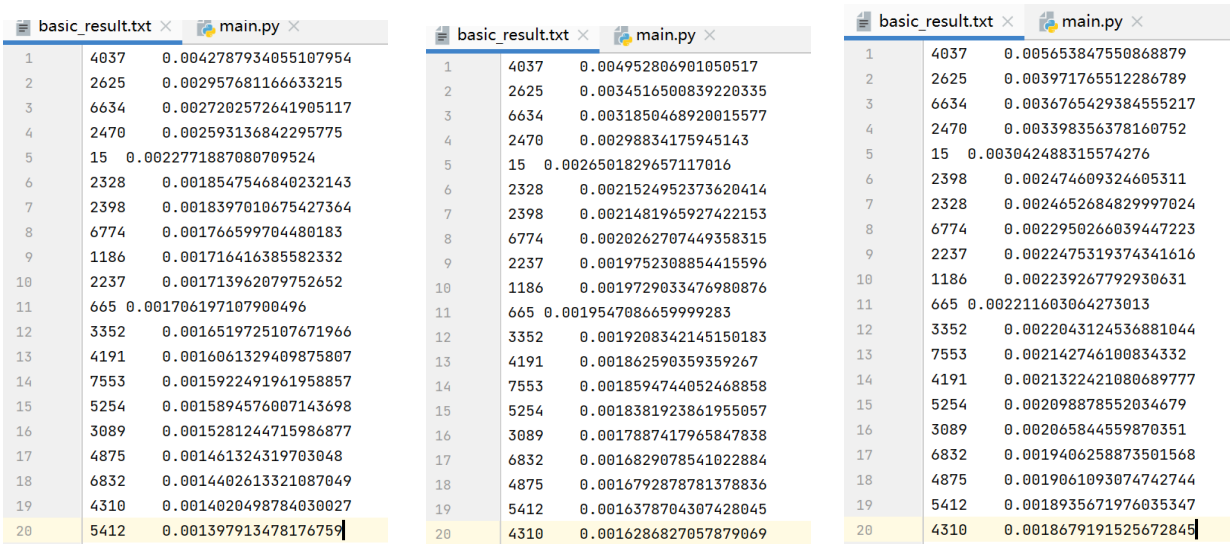
与本实验 Basic PageRank 算法的输出结果相比，从总体上看，调用库函数输出的 score 排名靠前的节点的得分更大一些（但排名第一的节点库函数得分比 Basic PageRank 算法低），一些节点的得分排名有所变化。

在得分排名前 100 的节点中，使用 Basic PageRank 算法与使用库函数的有 17 个不同的节点。

4.7.2 不同阻尼因子

对比设置不同阻尼因子时 score 结果的差异。图 4.2 是阻尼因子 d 分别为 0.75、0.85、0.95 时的排名最高的 20 个节点及其得分。

由图 4.2 可以看出，在不同阻尼因子下，这些节点的排名并没有太大差异。但随着阻尼因子的增大，score 排名比较靠前的节点（至少在前 100 个节点中）的 score 明显增大。



basic_result.txt × main.py ×

1	4037	0.0042787934055107954
2	2625	0.002957681166633215
3	6634	0.0027202572641905117
4	2470	0.002593136842295775
5	15	0.0022771887080709524
6	2328	0.0018547546840232143
7	2398	0.0018397010675427364
8	6774	0.001766599704480183
9	1186	0.001716416385582332
10	2237	0.001713962079752652
11	665	0.001706197107900496
12	3352	0.0016519725107671966
13	4191	0.0016061329409875807
14	7553	0.0015922491961958857
15	5254	0.0015894576007143698
16	3089	0.0015281244715986877
17	4875	0.001461324319703048
18	6832	0.0014402613321087049
19	4310	0.0014020498784030027
20	5412	0.0013979134781767599

(a) d = 0.75

basic_result.txt × main.py ×

1	4037	0.004952806901050517
2	2625	0.0034516500839220335
3	6634	0.0031850468920015577
4	2470	0.00298834175945143
5	15	0.0026501829657117016
6	2328	0.0021524952373620414
7	2398	0.0021481965927422153
8	6774	0.0020262707449358315
9	2237	0.0019752308854415596
10	1186	0.0019729033476980876
11	665	0.0019547086659999283
12	3352	0.0019208342145150183
13	4191	0.001862590359359267
14	7553	0.0018594744052468858
15	5254	0.0018381923861955057
16	3089	0.0017887417965847838
17	6832	0.0016829078541022884
18	4875	0.0016792878781378836
19	5412	0.0016378704307428045
20	4310	0.0016286827057879069

(b) d = 0.85

basic_result.txt × main.py ×

1	4037	0.005653847550868879
2	2625	0.003971765512286789
3	6634	0.0036765429384555217
4	2470	0.003398356378160752
5	15	0.003042488315574276
6	2398	0.002474609324605311
7	2328	0.0024652684829997024
8	6774	0.0022950266039447223
9	2237	0.0022475319374341616
10	1186	0.002239267792930631
11	665	0.002211603064273013
12	3352	0.0022043124536881044
13	7553	0.002142746100834332
14	4191	0.0021322421080689777
15	5254	0.002098878552034679
16	3089	0.002065844559870351
17	6832	0.0019406258873501568
18	4875	0.0019061093074742744
19	5412	0.0018935671976035347
20	4310	0.0018679191525672845

(c) d = 0.95

图 4.2: 不同阻尼因子下的 score 排名（前 20）

5 Block-Stripe PageRank 算法

5.1 实验原理

Block-Stripe PageRank 算法是一种基于 PageRank 的加速算法，其主要思想是通过对原始网页图进行分块，然后分块计算每个 block 内部的 PageRank 值，最后再通过一个全局迭代过程来更新每个 block 的 PageRank 值，从而达到加速计算的目的。

5.2 参数设置

1. block 数目，设置为 20。在本实验中，我测试了不同 block 数目下的程序执行情况，执行时间略有不同，但输出结果并无差异。
2. 阻尼因子，设置为 0.85。
3. 收敛阈值。当 PageRank 值的变化不再超过这个给定的阈值时，认为 score 列表收敛，并将其作为最终结果。此实验中，设置收敛阈值为 0.00001。
4. 最大循环次数，表示在更新 score 列表时的最多循环多少轮的计算，设置为 50。在多次实验中，该 Basic PageRank 算法基本都在 20 轮循环以内达到收敛状态。

5.3 block 分割

Block-Stripe PageRank 算法可以对每个 block 独立地进行 PageRank 计算，节约内存，提高效率。“block 分割”步骤的重点在于节点和链接关系的保存、block 分割的方法。

5.3.1 节点和链接关系的保存

与 Basic PageRank 算法不同，Block-Stripe PageRank 算法需要进行分块保存数据、分块计算。在分块计算时，如果更新第 k 个块的节点 score（即只更新 score 的一部分），此过程通过遍历块 k 对应

的转移矩阵实现，所以在数据读取阶段，需要将块 k 的转移矩阵中不在块 k 范围内的节点 ID 删除。

因此，转移矩阵 M 中不能只存节点链向的节点列表，还应保存该节点的出链节点总数，作为更新 score 时的参数。

同样使用稀疏矩阵来存储 M，只存储有出链的节点，并使用字典来存储节点和链接关系，key 为出链节点的 ID，value 是一个含两个元素的列表，第一个元素为该节点的出链总数，第二个是该节点链接的所有节点 ID 列表。

5.3.2 block 分割

Data.txt 读取后，可以得到节点总数 node_num，根据设置的 block_num，计算出每个 block 需要存储的节点数目和最后一个 block 中实际存储的节点数目。

分割策略：第 k 个 block 中要存储链接到 [k * block_node_num, (1+k) * block_node_num) 范围的所有节点。下面是块分割的具体步骤：

1. n2n_dic 字典存储了所有节点及其链接关系。对于第 k 个 block，创建一个临时字典 block_n2n_dic，使用 copy.deepcopy() 函数对 n2n_dic 进行深拷贝。这里必须进行深拷贝，因为后面会更改复制后的节点字典，不能让它影响到原数据 n2n_dic 。
2. 遍历 block_n2n_dic 中的所有元素。
3. 每个元素的 value 中存储两个数据：节点的出链总数和链向节点 ID 列表。判断 value 中的第二个元素（链向节点 ID 列表）中的 ID 是否在 [k * block_node_num, (1+k) * block_node_num) 范围内，如果不在，把这个 ID 删除。
4. 在处理完 ID 之后，判断 value 中的链向节点 ID 列表是否为空，如果为空，删除这个键值对。
5. 遍历 block_n2n_dic 结束。
6. 将 block_n2n_dic 以 “block_k” 命名，存到本地。回到步骤 1，继续处理第 k + 1 个 block。

block 分割关键代码

```
    # 按节点 ID 排序
    n2n_dic = dict(sorted(n2n_dic.items(), key=operator.itemgetter(0)))
    print(f"block_num: {block_num},node_num:{node_num},
        block_node_num:{block_node_num}, final_block_node_num:{final_block_node_num}")
    # save matrix to local
    matrix_path = "block_pg/mid_res/block_"
    for i in range(0, block_num):
        # 深拷贝
        block_n2n_dic = copy.deepcopy(n2n_dic)
        for key in list(block_n2n_dic.keys()):
            # 更新链接节点 ID 列表
            block_n2n_dic[key][1] = list(
                filter(lambda idx: idx >= i * block_node_num and idx < (i + 1) *
                    block_node_num, block_n2n_dic[key][1]))
            if block_n2n_dic[key][1] == []:
                #print(f"key = {key} is blank block_{i}:del {block_n2n_dic[key]}")
                del block_n2n_dic[key]
```

```
        f = open(matrix_path + str(i), "wb")
        pkl.dump(block_n2n_dic, f)
        print(f"save block_{i}!")
        f.close()
```

5.4 score 计算

Block-Stripe PageRank 算法中，每轮更新 score 列表，都需要先从本地读取一个 block 中的数据，更新部分 score ，再读取一个 block，更新部分 score，直到读取完所有块。然后再进行归一化和加入阻尼因子的计算。

这里只详细介绍按块更新 score 列表的步骤，归一化和阻尼因子的计算方法与 basic PageRank 算法一致。

1. 将当前 score 列表复制一份，作为上一轮的得分，用于计算新旧两轮得分之差的绝对值之和，作为判断停止条件的指标。
2. 进入 block_num 轮循环，每次从本地读取第 i 个 block 中的数据到字典 block_n2n_dic。
3. 遍历 block_n2n_dic 中所有键值对。
4. 对当前键值对，遍历 value 的第二个元素（链向节点 ID 列表），对该列表中所有节点，更新 score（加上 score[key]/n，其中 score[key] 为当前键值对的出链节点的得分，n 为 value 的第一个元素，表示该节点的出链总数）。
5. 一次 block 更新结束。继续读取 i + 1 个 block，回到步骤 3，直到全部 block 都读取完毕。
6. 对 score 列表进行归一化处理，加入阻尼因子计算，计算两轮差异，判断 score 更新是否停止。

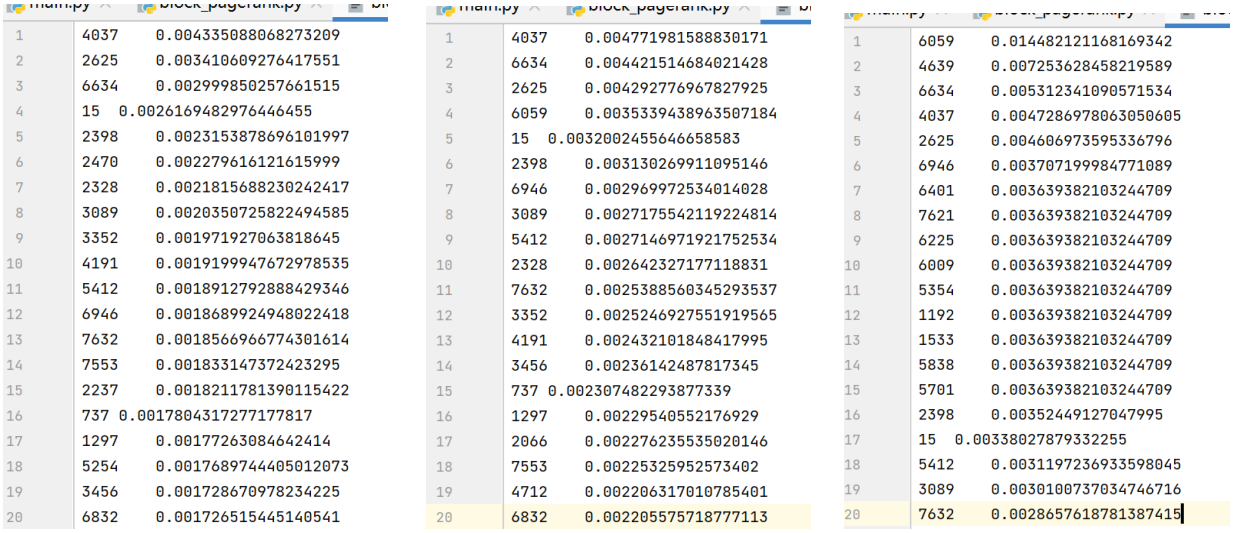
block score 更新关键代码

```
        last_score = score_ini.copy()
        # 分块计算
        for i in range(0, block_num):
            block_file = open("block_pg/mid_res/block_" + str(i), "rb")
            block_n2n_dic = pkl.load(block_file)
            block_file.close()
            print(f"load data block_{i}:{block_n2n_dic}")
            for key, value in block_n2n_dic.items():
                for to_node_idx in value[1]:
                    score_ini[to_node_idx] += last_score[key]/float(value[0])
```

5.5 结果分析

图 5.3 为不同阻尼因子情况下的输出结果，可以看出，这些节点的排名并没有太大差异。但随着阻尼因子的增大，score 排名比较靠前的节点（至少在前 100 个节点中）的 score 明显增大。

同时对比使用 Block PageRank 算法与使用库函数的输出结果，随着阻尼因子的增大，得分在前 100 中的节点差距变大。d = 0.75 时，有 6 个节点不同，d = 0.85 时，有 8 个节点不同，d = 0.9 时，有 18 个节点不同。



1	4037	0.004335088068273209
2	2625	0.003410609276417551
3	6634	0.002999850257661515
4	15	0.0026169482976446455
5	2398	0.0023153878696101997
6	2470	0.002279616121615999
7	2328	0.0021815688230242417
8	3089	0.0020350725822494585
9	3352	0.001971927063818645
10	4191	0.0019199947672978535
11	5412	0.0018912792888429346
12	6946	0.0018689924948022418
13	7632	0.0018566966774301614
14	7553	0.001833147372423295
15	2237	0.0018211781390115422
16	737	0.0017804317277177817
17	1297	0.00177263084642414
18	5254	0.0017689744405012073
19	3456	0.001728670978234225
20	6832	0.001726515445140541

(a) d = 0.75

1	4037	0.004771981588830171
2	6634	0.004421514684021428
3	2625	0.004292776967827925
4	6059	0.0035339438963507184
5	15	0.0032002455646658583
6	2398	0.003130269911095146
7	6946	0.002969972534014028
8	3089	0.0027175542119224814
9	5412	0.0027146971921752534
10	2328	0.002642327177118831
11	7632	0.0025388560345293537
12	3352	0.0025246927551919565
13	4191	0.002432101848417995
14	3456	0.00236142487817345
15	737	0.002307482293877339
16	1297	0.00229540552176929
17	2066	0.002276235535020146
18	7553	0.00225325952573402
19	4712	0.002206317010785401
20	6832	0.002205575718777113

(b) d = 0.85

1	6059	0.014482121168169342
2	4639	0.007253628458219589
3	6634	0.005312341090571534
4	4037	0.0047286978063050605
5	2625	0.004606973595336796
6	6946	0.003707199984771089
7	6401	0.003639382103244709
8	7621	0.003639382103244709
9	6225	0.003639382103244709
10	6009	0.003639382103244709
11	5354	0.003639382103244709
12	1192	0.003639382103244709
13	1533	0.003639382103244709
14	5838	0.003639382103244709
15	5701	0.003639382103244709
16	2398	0.00352449127047995
17	15	0.00338027879332255
18	5412	0.0031197236933598045
19	3089	0.0030100737034746716
20	7632	0.0028657618781387415

(c) d = 0.9

图 5.3: Block-Stripe PageRank 算法，不同阻尼因子下的 score 排名（前 20）