

《数据科学导论》PageRank 实验报告 (DS-Lab-PageRank)

中国人民大学 信息学院 崔冠宇 2018202147

备注: 我的实验环境: Mac OS X El Captain (10.11.6) + Python 3.8.2.

一、实验题目: PageRank——算法实现与理论证明.

二、实验目的:

- 掌握基本 PageRank 算法的 python 编程实现;
- 掌握个性化 PageRank 算法的 python 编程实现;
- 会证明有关个性化 PageRank 线性可加性的题目.

三、实验方法:

在 macOS 环境下, 编写 python 语言程序 (详见末尾“程序清单”部分), 实现作业相关要求.

程序编写完毕后, 在测试集上运行 python 文件, 并观察程序输出. 如果程序结果与我们的预期相符合, 则将其应用到较大规模的数据集上; 如果程序有问题, 则仔细分析原因, 查找资料, 修改程序, 直至正确为止.

最后使用数学方法证明 Personalized PageRank 的线性可加性.

四、PageRank 函数实现思路

本次实验首先要解决的问题就是: 如何存储稀疏图? 回忆上学期学过的《数据结构与算法》课程, 我们曾经用邻接表来存储图, 考虑到 PageRank 的算法, 我决定使用下面的结构来保存图:

```
1 # Graph = {'node' : [ [inEdge], outDegree ]}
```

即图是一个字典, 键 (key) 是顶点字符串, 值 (value) 是一个列表: 列表中第一项是该节点的入边表, 用于 PageRank 算法中计算各入边的贡献; 列表中第二项是该节点的出度.

下面主要介绍我的思路, 具体代码实现请参见末尾的“程序清单”部分的 **2018202147-PageRank-Test.py**.

1. 首先打开文件, 将文件内容读入列表中, 供下一步建图使用;
2. 读取每行内容, 将每条边前面的顶点增加至后面的顶点的入边表, 并将前面顶点的出度增加一, 就建立了我们需要的图;
3. 若图是空图, 直接返回空列表; 否则记录下悬挂点 (dangling_nodes), 用于后续处理.
4. 为了加快迭代, 我建立了一个图字典键值到权重列表索引下标的字典. 迭代时仅拷贝权重列表, 可能可以加快速度.

5. 然后设置权值初值为均匀分布, 开始迭代. 由于悬挂点与每个点之间都加了一条边, 我们可以不在原图上修改, 而是首先计算悬挂点的权重总和 `dangling_sum`, 在迭代过程中在计算入边的贡献时, 给每个节点都增加 `dangling_sum / graph_size`, 即迭代时有

$$\begin{aligned} \mathbf{p}^{(n+1)}[i] &= \frac{1 - \alpha}{\text{graph_size}} + \alpha \sum_{j \rightarrow i} \frac{\mathbf{p}^{(n)}[j]}{\text{out_degree}[j]} \\ &= \frac{1 - \alpha}{\text{len}(\text{graph})} + \alpha \left(\sum_{j \in \text{graph}[i][0]} \frac{\mathbf{p}^{(n)}[j]}{\text{graph}[j][1]} + \frac{\sum_{k \in \text{dangling_nodes}} \mathbf{p}^{(n)}[k]}{\text{len}(\text{graph})} \right) \end{aligned}$$

6. 每迭代一次, 计算一次迭代向量之间的差距 (曼哈顿距离):

$$\begin{aligned} \text{delta} &= \text{graph_size} \cdot \|\mathbf{p}^{(n+1)} - \mathbf{p}^{(n)}\|_1 \\ &= \text{graph_size} \cdot \sum_i |\mathbf{p}^{(n+1)}[i] - \mathbf{p}^{(n)}[i]| \end{aligned}$$

当 $\text{delta} < 1\text{e-}6 \times \text{len}(\text{graph})$ 时, 认为收敛, 停止迭代.

7. 迭代完成后, 将权值与原字典的键值拼接为新字典, 排序后返回顶点名称列表.

五、Personalized PageRank 函数实现思路

Personalized PageRank 与普通 PageRank 有两个主要区别: 一是权值初始化与普通 PageRank 的默认均匀分布不同, 需要按照用户自定的权值作为初值; 二是迭代时的式子不同, 变为了

$$\begin{aligned} \mathbf{p}^{(n+1)}[i] &= (1 - \alpha)\mathbf{p}^{(0)}[i] + \alpha \sum_{j \rightarrow i} \frac{\mathbf{p}^{(n)}[j]}{\text{out_degree}[j]} \\ &= (1 - \alpha)\mathbf{p}^{(0)}[i] + \alpha \left(\sum_{j \in \text{graph}[i][0]} \frac{\mathbf{p}^{(n)}[j]}{\text{graph}[j][1]} + \frac{\sum_{k \in \text{dangling_nodes}} \mathbf{p}^{(n)}[k]}{\text{len}(\text{graph})} \right) \end{aligned}$$

六、实验结果 (包含与 NetworkX 包的速度对比)

在进行测试之前, 我们需要对较大的图数据进行预处理. 总体思路很简单, 用 `python` 读入每一行的数据, 存入列表中, 然后丢弃前几行数据描述, 并且对每一行数据字符串的 `TAB` 替换为逗号, 最后输出即可. 除此之外, 还需要生成种子文件. 我将这些代码写到了 `preprocess.py` 中, 详细代码请见末尾的“程序清单”部分.

首先运行 `preprocess.py`, 获得处理后的图文件以及种子文件. 下面展示较大规模的数据集上 PageRank 和 Personalized PageRank 前十名的顶点与分值:

```
1. CuiGuanyu@Mac-mini: ~/Desktop/PageRank (zsh)
CuiGuanyu@Mac-mini ~ ➤ cd Desktop/PageRank
CuiGuanyu@Mac-mini ~/Desktop/PageRank ➤ python3 2018202147-PageRank-Test.py
Time used for PageRank: 2.736 s.
Node      Score
18         0.00465719
737        0.00288068
1719       0.00215456
790        0.00212152
118        0.00205882
136        0.00203518
143        0.00200368
40         0.00159238
1619       0.00152633
4415       0.00147818

Time used for PPR: 2.850 s.
Node      Score
18         0.00728587
31         0.00640814
27         0.00604428
34         0.00589356
30         0.00587523
40         0.00575013
0          0.00557235
12         0.00536656
1          0.00532993
28         0.00500148
CuiGuanyu@Mac-mini ~/Desktop/PageRank ➤
```

可见两函数运行时间均为 3s 左右, 除此之外我在能显示运行时占用内存的软件上运行时, 发现最大占用内存 <80 MiB. 我还写了一个与 NetworkX 库对比的脚本, 具体代码请见 `nx-pr-compare.py`, 下面展示对比结果:

```

1. CuiGuanyu@Mac-mini: ~/Desktop/PageRank (zsh)
CuiGuanyu@Mac-mini ~ > cd Desktop/PageRank
CuiGuanyu@Mac-mini ~/Desktop/PageRank > python3 nx_pr_compare.py
Time used for nx PageRank: 10.101 s.
Node      Score
18         0.00465719
737        0.00288068
1719       0.00215456
790        0.00212152
118        0.00205882
136        0.00203518
143        0.00200368
40         0.00159238
1619       0.00152633
4415       0.00147818

Time used for nx PPR: 9.914 s.
Node      Score
18         0.00728587
31         0.00640814
27         0.00604428
34         0.00589356
30         0.00587523
40         0.00575013
0          0.00557235
12         0.00536656
1          0.00532993
28         0.00500148
CuiGuanyu@Mac-mini ~/Desktop/PageRank >

```

可见两函数运行时间均为 10s 左右, 除此之外我在能显示运行时占用内存的软件上运行时, 发现最大占用内存 >600 MiB.

七、大规模图数据的应对

后续我们可能要考虑如何应对超大规模的图数据, 我有以下几点想法:

1. 当图很大时, 可以考虑适当修改算法的细节, 使得能够并行计算.
 - 对某些不连通而且连通分支数目比较多的图, 可以把不同的连通分量分配给集群中不同的机器进行并行计算 (与我们之前学过的 MapReduce 有关的知识联系了起来), 进而加快我们的算法. 我设想的一种多连通分支的稀疏图的 MapReduce PageRank 算法框架如下:

Algorithm 多连通分支稀疏图 MapReduce PageRank

输入: 图的邻接矩阵 \mathbf{A}

计算出度矩阵 \mathbf{M} , $\mathbf{M}_{ii} = \sum_j \mathbf{A}_{ij}$, $i = 1, 2, \dots, n$

因为图不连通, 将 \mathbf{A} 的行调整顺序使之分块. \mathbf{M} 按同样方式调整

Split: 将 \mathbf{A}, \mathbf{M} 按不同的块分割, 交给 Mapper

Map: $\langle \text{ComponentID}, \langle \mathbf{A}_i, \mathbf{M}_i \rangle \rangle \xrightarrow{\text{PageRank 算法}} \text{List}(\langle \text{NodeID}, \text{Score} \rangle)$

Shuffle: 按 Score 进行排序

Reduce: 聚合数据, 输出

输出: 迭代完成后的向量 \mathbf{p}

- 考虑到现实中网络的“蝴蝶结”结构, 大多数网页位于蝴蝶结 (最大的弱连通分支) 上, 使得我们应该更加关注大规模连通图的 PageRank 优化.

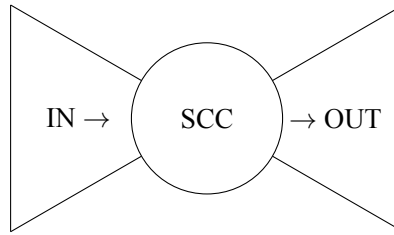


图 1: 网络“蝴蝶结”的主要结构

比如我们可以考虑用稀疏矩阵存储大规模的图, 然后实现稀疏矩阵的乘法¹, 每一次迭代都利用矩阵运算的并行算法, 将矩阵乘法的任务分配给集群中的机器进行并行计算, 然后将各部分结果汇总, 完成一次迭代.

2. 另, 我在知乎上找到了一些分布式计算 PageRank 的代码/算法框架, 附在文后.
3. 当图很大的时候, 迭代时会涉及到绝对值较小的数的计算, 而这容易导致计算结果不准确, 需要我们适当改进算法, 避免这种情况发生.

八、证明题

题目: 现有全连通的平凡图 $G = \langle V, E \rangle$, $|V| = n$, n 个顶点 (包含种子) 的分数向量 $p^{(0)} = [\lambda_1, \lambda_2, \dots, \lambda_n]$, 满足 $\sum_{i=1}^n \lambda_i = 1$, 以及 Personalized PageRank 函数 $PPR(p^{(0)}, G, \alpha)$. 函数 PPR 输出的是所有节点的最终收敛分数向量 p^* , 即 $p^* \leftarrow PPR(p^{(0)}, G, \alpha)$. 令 $p_i^{(0)} = [0, 0, \dots, 1, \dots, 0]$ (长度和 $p^{(0)}$ 一样, 但是第 i 个元素为 1, 其余全为 0), 即

$$p_i^{(0)}[j] = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{if } j \neq i. \end{cases}, 1 \leq j \leq n.$$

再令 $p_i^* = PPR(p_i^{(0)}, G, \alpha)$. 求证: $p^* = \sum_{i=1}^n \lambda_i p_i^*$.

¹我在上面实现的算法不是严格意义的矩阵乘法, 而是使用的邻接表做的迭代.

证明:

证法一.

先用数学归纳法证明一个命题: $\mathbf{p}^{(m)} = \sum_{i=1}^n \lambda_i \mathbf{p}_i^{(m)} (m \in \mathbb{N})$.

① 当 $m = 0$ 时, $\mathbf{p}^{(0)} = [\lambda_1, \lambda_2, \dots, \lambda_n] = \sum_{i=1}^n \lambda_i \mathbf{p}_i^{(0)}$.

② 假定 $m = k$ 时命题成立, 即 $\mathbf{p}^{(k)} = \sum_{i=1}^n \lambda_i \mathbf{p}_i^{(k)}$. 则当 $m = k+1$ 时, 由于 $\mathbf{p}_i^{(k+1)} = \alpha \mathbf{p}_i^{(k)} \mathbf{L} + (1-\alpha) \mathbf{p}_i^{(0)} (i = 1, 2, \dots, n)$, 所以 $\sum_{i=1}^n \lambda_i \mathbf{p}_i^{(k+1)} = \alpha (\sum_{i=1}^n \lambda_i \mathbf{p}_i^{(k)}) \mathbf{L} + (1-\alpha) (\sum_{i=1}^n \lambda_i \mathbf{p}_i^{(0)}) = \alpha \mathbf{p}^{(k)} \mathbf{L} + (1-\alpha) \mathbf{p}^{(0)} = \mathbf{p}^{(k+1)}$, 即命题对 $m = k+1$ 成立. 综上, 命题对 $\forall m \in \mathbb{N}$ 都成立.

同时对两侧取极限, 即得结论. \square

证法二.

先写出迭代式: $\mathbf{p}^{(m+1)} = \alpha \mathbf{p}^{(m)} \mathbf{L} + (1-\alpha) \mathbf{p}^{(0)}$. 记 \mathbf{p}_i^* 是 $\mathbf{p}_i^{(0)}$ 作为起始向量迭代至收敛的结果, 则按照迭代收敛的不动点定义有 $\mathbf{p}_i^* = \alpha \mathbf{p}_i^* \mathbf{L} + (1-\alpha) \mathbf{p}_i^{(0)} (i = 1, 2, \dots, n)$. 将上述含有 \mathbf{p}_i 的式子分别乘 λ_i 然后相加, 得

$$\sum_{i=1}^n \lambda_i \mathbf{p}_i^* = \alpha (\sum_{i=1}^n \lambda_i \mathbf{p}_i^*) \mathbf{L} + (1-\alpha) \mathbf{p}^{(0)}.$$

同时注意到对于 \mathbf{p}^* 有

$$\mathbf{p}^* = \alpha \mathbf{p}^* \mathbf{L} + (1-\alpha) \mathbf{p}^{(0)}.$$

因为递推式 (形如 $x_{n+1} = ax_n + b$) 是关于 \mathbf{p} 的一次式, 根据特征方程是一次的, 若有不动点一定是唯一的, 故对比两式有 $\mathbf{p}^* = \sum_{i=1}^n \lambda_i \mathbf{p}_i^*$. \square

九、实验小结

在本次实验中, 我有以下收获:

1. 手动实现了 PageRank 算法以及 Personalized PageRank 算法, 加强了对 PageRank 的理解, 深化了记忆.
2. 通过将我的函数与 NetworkX 库的代码对比, 我的代码在这种条件下快于 NetworkX 的 PageRank. (但是实际上我没有 NetworkX 的通用性, 这么比较有些不合适.)
3. 联系之前所学的分布式计算的知识, 思考了应对大规模数据的可能解决方法.
4. 通过证明有关 Personalized PageRank 线性可加性的题目, 更进一步理解了算法背后的数学原理.

十、程序清单

1. PageRank 及 Personalized PageRank 测试版代码 (为了方便显示数据, 返回值与提交版略不同, 但是主体算法一致) —— **2018202147-PageRank-Test.py**:

```
1 #2018202147-PageRank.py
```

```
2
```

```

3 #@param input_file_name: 描述一个graph的纯文本邻接表文件名,如' E:\graph.txt' .
4 #@param damping_factor
5 def PageRank( input_file_name , damping_factor ):
6     # graph = {'node': [[inEdge], outDegree]}
7     graph = {}
8
9     # Open and read file.
10    with open(input_file_name, 'r') as f:
11        contents = f.readlines()
12    # Create adjacency list of a graph.
13    for line in contents:
14        # if line.strip() == '':
15            #continue
16        left, right = line.split(',')
17        right = right.strip()
18        # Append inEdge
19        graph.setdefault(right,[ [], 0 ])[0].append(left)
20        # Increase outEdge
21        graph.setdefault(left,[ [], 0 ])[1] += 1
22    f.close()
23    del line, contents
24    del left, right
25
26    # Graph size
27    graph_size = len(graph)
28    # Empty graph
29    if graph_size == 0:
30        return []
31
32    # Collect dangling nodes, but don't add real edge in the graph.
33    # Instead, calculate dangling_sum below.
34    dangling_nodes = [node for node in graph.keys()\
35        if graph[node][1] == 0]
36
37    # Dict from nodename(key) to list index, avoid copying dicts.
38    # e.g. :
39    # graph_dict = { 'a':... , 'b':..., 'c':... }

```

```

40     # weight_list = [ 0.33, 0.33, 0.33 ]
41     # Only iterate weight_list.
42     node_to_index = dict(zip(graph.keys(), range(graph_size)))
43
44     # Initial weight
45     old_weight = [1.0 / graph_size for i in range(graph_size)]
46
47     # Tolerance of errorness.
48     tol = 1e-6
49     i = 0
50     # Iterate
51     while True:
52         # Dangling sum from dangling nodes to every node.
53         dangling_sum = 0
54         for node in dangling_nodes:
55             dangling_sum += old_weight[node_to_index[node]]
56         # Base : (1-alpha) / n
57         new_weight = [\
58             (1.0 - damping_factor) / graph_size \
59             for i in range(graph_size)]
60
61         # Dangling nodes contribute to other nodes.
62         # alpha * dangling_sum / graph_size
63         dangling_avg = damping_factor / graph_size * dangling_sum;
64         # Iterate nodes.
65         for node in graph.keys():
66             # Iterate innodes.
67             for innode in graph[node][0]:
68                 # Add weights.
69                 new_weight[node_to_index[node]] += \
70                     damping_factor / graph[innode][1] \
71                     * old_weight[node_to_index[innode]]
72                 new_weight[node_to_index[node]] += dangling_avg
73
74         delta = sum([abs(new_weight[i] - old_weight[i])\
75                     for i in range(graph_size)])
76         if delta < tol * graph_size:

```



```

77         break
78     # Update.
79     old_weight = new_weight
80     i += 1
81 del tol, i
82
83 # Sort.
84 sorted_result = \
85     sorted( dict(zip(graph.keys(), new_weight)).items(), \
86             key = lambda i: (i[1], i[0]), \
87             reverse = True )
88 # Generate list and score.
89 # Comment 'scores' in homework code.
90 node_list_in_descending_order = [term[0] for term in sorted_result]
91 scores = [term[1] for term in sorted_result]
92 return node_list_in_descending_order , scores
93
94
95 #@param input_Graph: 描述一个graph的纯文本邻接表文件名,如' E:\graph.txt'。
96 #@param input_Seed: 描述一个种子集的纯文本文件名,如' E:\seed.txt'。
97 #@param damping_factor
98 def PPR( input_Graph , input_Seed , damping_factor ) :
99     # Read graph like PageRank(...).
100     # graph = {'node': [[inEdge], outDegree]}
101     graph = {}
102
103     # Open and read file.
104     with open(input_Graph, 'r') as f:
105         contents = f.readlines()
106     # Create adjacency list of a graph.
107     for line in contents:
108         # if line.strip() == '':
109             #continue
110         left, right = line.split(',')
111         right = right.strip()
112         # Append inEdge
113         graph.setdefault(right,[ [], 0 ])[0].append(left)

```

```

114         # Increase outEdge
115         graph.setdefault(left,[ [] , 0 ])[1] += 1
116     f.close()
117
118     # Graph size
119     graph_size = len(graph)
120     # Empty graph
121     if graph_size == 0:
122         return []
123
124     # Read seed.
125     # seed = {'node' : weight}
126     seed = {}
127     with open(input_Seed, 'r') as f:
128         contents = f.readlines()
129     # Create adjacency list of a graph.
130     for line in contents:
131         # if line.strip() == '':
132             #continue
133         left, right = line.split(',')
134         weight = float(right)
135         # Add weight.
136         seed[left] = weight
137     f.close()
138     del line, contents
139     del left, right, weight
140
141     # Collect dangling nodes, but don't add real edge in the graph.
142     # Instead, calculate dangling_sum below.
143     dangling_nodes = [node for node in graph.keys()\
144         if graph[node][1] == 0]
145
146     # Dict from nodename(key) to list index, avoid copying dicts.
147     # e.g. :
148     # graph_dict = { 'a':... , 'b':..., 'c':... }
149     # weight_list = [ 0.33, 0.33, 0.33 ]
150     # Only iterate weight_list.

```

```

151     node_to_index = dict(zip(graph.keys(), range(graph_size)))
152
153     # Initial weight using seed
154     p0 = [0 for _ in range(graph_size)]
155     for node in seed.keys():
156         p0[node_to_index[node]] = seed[node]
157     old_weight = p0
158
159     # Tolerance of errorness.
160     tol = 1e-6
161     i = 0
162     # Iterate
163     while True:
164         # Dangling sum from dangling nodes to every node.
165         dangling_sum = 0
166         for node in dangling_nodes:
167             dangling_sum += old_weight[node_to_index[node]]
168         # Base : (1-alpha) * p0
169         new_weight = [\
170             (1.0 - damping_factor) * p0[i]\
171             for i in range(graph_size)]
172
173         # Dangling nodes contribute to other nodes.
174         # alpha * dangling_sum / graph_size
175         dangling_avg = damping_factor / graph_size * dangling_sum;
176         # Iterate nodes.
177         for node in graph.keys():
178             # Iterate innodes.
179             for innode in graph[node][0]:
180                 # Add weights.
181                 new_weight[node_to_index[node]] += \
182                     damping_factor / graph[innode][1] \
183                     * old_weight[node_to_index[innode]]
184                 new_weight[node_to_index[node]] += dangling_avg
185
186         delta = sum([abs(new_weight[i] - old_weight[i])\
187                     for i in range(graph_size)])

```

```

188         if delta < tol * graph_size:
189             break
190         # Update.
191         old_weight = new_weight
192         i += 1
193     del tol, i
194
195     # Sort.
196     sorted_result = \
197         sorted( dict(zip(graph.keys(), new_weight)).items(), \
198             key = lambda i: (i[1], i[0]), \
199             reverse = True )
200     # Generate list and score.
201     # Comment 'scores' in homework code.
202     node_list_in_descending_order = [term[0] for term in sorted_result]
203     scores = [term[1] for term in sorted_result]
204     return node_list_in_descending_order , scores
205
206 # Test PR.
207 import time
208 time_start = time.time()
209 result = PageRank('soc-Epinions1_processed.txt', 0.85)
210 result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
211 time_end = time.time()
212 print( 'Time used for PageRank: %.3f s.' %(time_end - time_start) )
213 print('Node\t\tScore')
214 for k, v in result_dict_rank10.items():
215     print('%s\t\t%.8f' % (k, v))
216
217 print()
218
219 # Test PPR.
220 time_start = time.time()
221 result = PPR('soc-Epinions1_processed.txt', 'soc-Epinions1_seed.txt', 0.85)
222 result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
223 time_end = time.time()
224 print( 'Time used for PPR: %.3f s.' %(time_end - time_start) )

```

```

225 print('Node\t\tScore')
226 for k, v in result_dict_rank10.items():
227     print('%s\t\t%.8f' % (k, v))

```

2. 数据预处理——**preprocess.py**:

```

1 # Preprocess data.
2
3 # Prepare graph
4 def prepare_graph():
5     f_ori = open('soc-Epinions1.txt', 'r')
6     contents = f_ori.readlines()
7     f_ori.close()
8
9     contents = contents[4:]
10    contents = [line.replace('\t', ',') for line in contents]
11    f_pre = open('soc-Epinions1_processed.txt', 'w')
12
13    for line in contents:
14        f_pre.write(line)
15    f_pre.close()
16
17 # Generate seed.
18 def generate_seed():
19     f_seed = open('soc-Epinions1_seed.txt', 'w')
20     for i in range(50):
21         f_seed.write(str(i) + ',0.02\n')
22     f_seed.close()
23
24 prepare_graph()
25 generate_seed()

```

3. 与 nx 库对比——**nx_pr_compare.py**:

```

1 import networkx as nx
2
3 def nx_PR_benchmark(input_file_name , damping_factor):
4     G = nx.DiGraph()
5     edges = []

```

```

6     # Open and read file.
7     with open(input_file_name, 'r') as f:
8         contents = f.readlines()
9     # Create adjacency list of a graph.
10    for line in contents:
11        # if line.strip() == '':
12            #continue
13        left, right = line.split(',')
14        right = right.strip()
15        # Append inEdge
16        edges.append((left, right))
17    f.close()
18    G.add_edges_from(edges)
19
20    sorted_result = sorted(\
21        nx.pagerank(G, alpha = damping_factor).items(), \
22        key = lambda x: (x[1], x[0]), reverse = True)
23
24    # Generate list and score.
25    # Comment 'scores' in homework code.
26    node_list_in_descending_order = [term[0] for term in sorted_result]
27    scores = [term[1] for term in sorted_result]
28    return node_list_in_descending_order , scores
29
30 def nx_PPR_benchmark(input_Graph, input_Seed, damping_factor):
31     G = nx.DiGraph()
32     nodes = {}
33     edges = []
34     # Open and read file.
35     with open(input_Graph, 'r') as f:
36         contents = f.readlines()
37     # Create adjacency list of a graph.
38     for line in contents:
39         # if line.strip() == '':
40             #continue
41         left, right = line.split(',')
42         right = right.strip()

```

```

43         # Append inEdge
44         edges.append((left, right))
45         # Add node name.
46         nodes[left] = 0
47         nodes[right] = 0
48     f.close()
49     G.add_edges_from(edges)
50
51     # Read seed.
52     # seed = {'node' : weight}
53     seed = {}
54     with open(input_Seed, 'r') as f:
55         contents = f.readlines()
56     # Create adjacency list of a graph.
57     for line in contents:
58         # if line.strip() == '':
59             #continue
60         left, right = line.split(',')
61         weight = float(right)
62         # Add weight.
63         seed[left] = weight
64     f.close()
65
66     nstart = nodes
67     for node in seed:
68         nstart[node] = seed[node]
69
70     dangling = dict.fromkeys(nodes, 1 / len(nodes))
71     sorted_result = sorted(\
72         nx.pagerank(G, alpha = damping_factor, \
73             personalization = seed, nstart = nstart, \
74             dangling = dangling).items(), \
75         key = lambda x: (x[1], x[0]), reverse = True)
76
77     # Generate list and score.
78     # Comment 'scores' in homework code.
79     node_list_in_descending_order = [term[0] for term in sorted_result]

```

```

80     scores = [term[1] for term in sorted_result]
81     return node_list_in_descending_order , scores
82
83     return None
84
85
86 # Test nx PR.
87 import time
88 time_start = time.time()
89 result = nx_PR_benchmark('soc-Epinions1_processed.txt', 0.85)
90 result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
91 time_end = time.time()
92 print( 'Time used for nx PageRank: %.3f s.' %(time_end - time_start) )
93 print('Node\t\tScore')
94 for k, v in result_dict_rank10.items():
95     print('%s\t\t%.8f' % (k, v))
96
97 print()
98
99 # Test nx PPR.
100 time_start = time.time()
101 result = nx_PPR_benchmark('soc-Epinions1_processed.txt', 'soc-Epinions1_seed.
    txt', 0.85)
102 result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
103 time_end = time.time()
104 print( 'Time used for nx PPR: %.3f s.' %(time_end - time_start) )
105 print('Node\t\tScore')
106 for k, v in result_dict_rank10.items():
107     print('%s\t\t%.8f' % (k, v))

```


4. 有知乎用户²用 Spark 实现的分布式计算 PageRank:

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
}
```

5. PageRank 算法并行实现, <https://blog.csdn.net/garfielder007/article/details/50889083>

²@ 严林. <https://www.zhihu.com/question/29213275/answer/43566160>, 以及同问题下面也有 @ 思哲回答了 hadoop 框架下 MapReduce 的算法, 此处略.