# 《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:

```
# 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. 迭代时仅拷贝权值列表, 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

$$
\begin{aligned}
\mathbf{p}^{(n+1)}[i] &= \frac{1-\alpha}{\text{graph\_size}} + \alpha \sum_{j \to i} \frac{\mathbf{p}^{(n)}[j]}{\text{out\_degree}[j]} \\
&= \frac{1-\alpha}{\text{len(graph)}} + \alpha \Big( \sum_{j \in \text{graph[i][0]}} \frac{\mathbf{p}^{(n)}[j]}{\text{graph}[j][1]} + \frac{\sum_{k \in \text{dangling\_nodes}} \mathbf{p}^{(n)}[k]}{\text{len(graph)}} \Big)
\end{aligned}
$$

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 delta $<$ 1e-6 $\times$ len(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

$$
\begin{aligned}
\mathbf{p}^{(n+1)}[i] &= (1-\alpha)\mathbf{p}^{(0)}[i] + \alpha \sum_{j \to i} \frac{\mathbf{p}^{(n)}[j]}{\text{out\_degree}[j]} \\
&= (1-\alpha)\mathbf{p}^{(0)}[i] + \alpha \Big( \sum_{j \in \text{graph[i][0]}} \frac{\mathbf{p}^{(n)}[j]}{\text{graph}[j][1]} + \frac{\sum_{k \in \text{dangling\_nodes}} \mathbf{p}^{(n)}[k]}{\text{len(graph)}} \Big)
\end{aligned}
$$

## 六、实验结果 (包含与 NetworkX 包的速度对比)

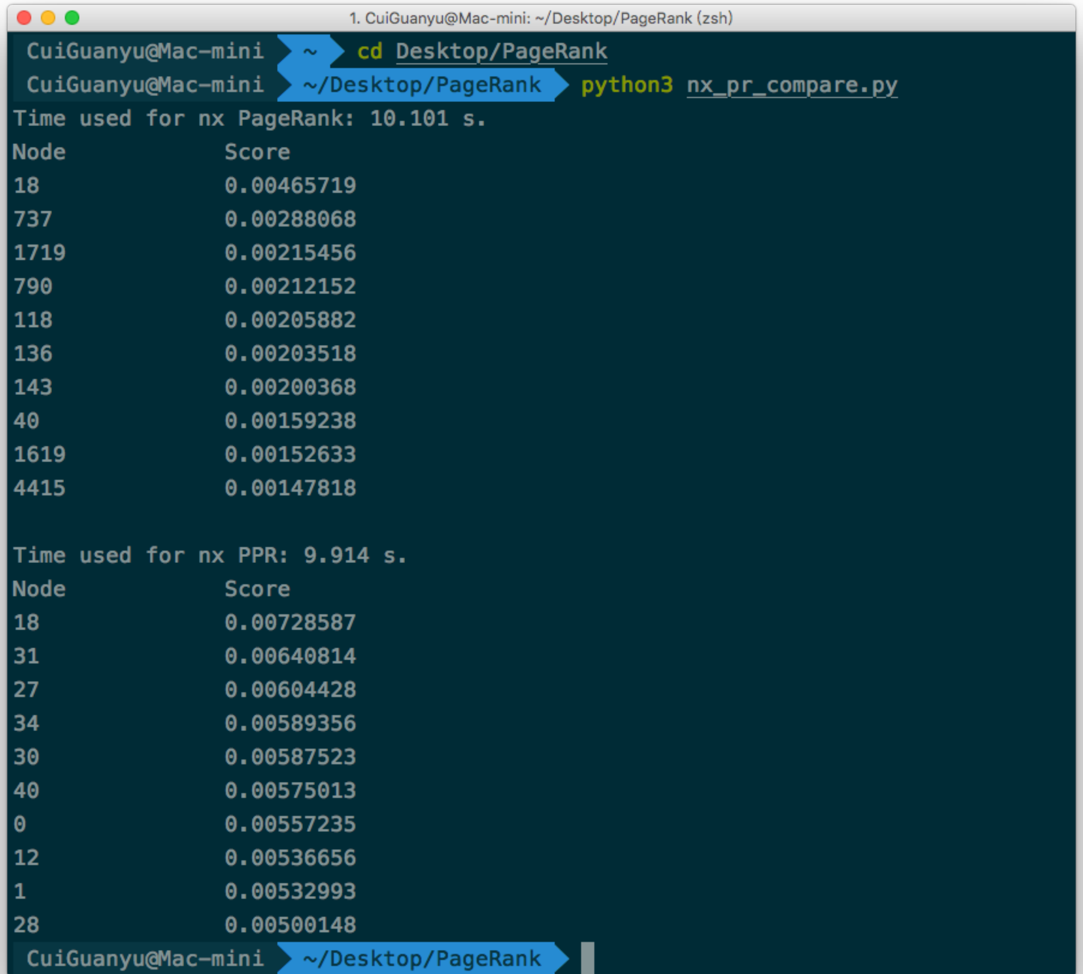
在进行测试之前, 我们需要对较大的图数据进行预处理. 总体思路很简单, 用 python 读入每一行的数据, 存入列表中, 然后丢弃前几行数据描述, 并且对每一行数据字符串的 TAB 替换为逗号, 最后输出即可. 除此之外, 还需要生成种子文件. 我将这些代码写到了 **preprocess.py** 中, 详细代码请见末尾的"程序清单"部分.

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:




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 $\mathbf{M}$, $\mathbf{M}_{ii} = \sum_j \mathbf{A}_{ij}, i = 1, 2, \ldots, n$

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

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.

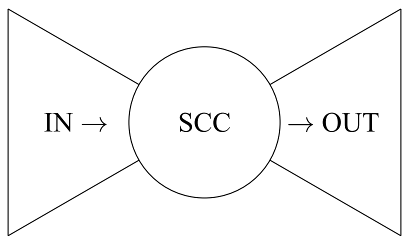



**图** 1: 网络"蝴蝶结"的主要结构

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 $G =< V, E >, |V| = n$, $n$ 个顶点 (包含种子) 的分数向量 $p^{(0)} = [\lambda_1, \lambda_2, \ldots, \lambda_n]$, 满足 $\sum\limits_{i=1}^{n} \lambda_i = 1$, 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 $p_i^{(0)} = [0, 0, \ldots, 1, \ldots, 0]$ (长度和 $p^{(0)}$ 一样, 但是第 $i$ 个元素为 1, 其余全为 0), 即

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

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

[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: $\mathbf{p}^{(m)}=\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^{(m)}(m\in\mathbb{N})$.

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

② 假定 $m=k$ 时命题成立, 即 $\mathbf{p}^{(k)}=\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^{(k)}$. 则当 $m=k+1$ 时, 由于 $\mathbf{p}_i^{(k+1)}=\alpha\mathbf{p}_i^{(k)}\mathbf{L}+(1-\alpha)\mathbf{p}_i^{(0)}(i=1,2,\cdots,n)$, 所以 $\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^{(k+1)}=\alpha(\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^{(k)})\mathbf{L}+(1-\alpha)(\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^{(0)})=\alpha\mathbf{p}^{(k)}\mathbf{L}+(1-\alpha)\mathbf{p}^{(0)}=\mathbf{p}^{(k+1)}$, 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. □

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 $\mathbf{p}_i^*=\alpha\mathbf{p}_i^*\mathbf{L}+(1-\alpha)\mathbf{p}_i^{(0)}(i=1,2,\ldots,n)$. 将上述含有 $\mathbf{p}_i$ 的式子分别乘 $\lambda_i$ 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 $\mathbf{p}^*=\sum\limits_{i=1}^{n}\lambda_i\mathbf{p}_i^*$. □

## 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:**

```
#2018202147-PageRank.py

```

```
#@param input_file_name: 描述一个graph的纯文本邻接表文件名,如’E:\graph.txt’.
#@param damping_factor
def PageRank( input_file_name , damping_factor ):
    # graph = {'node': [[inEdge], outDegree]}
    graph = {}

    # Open and read file.
    with open(input_file_name, 'r') as f:
        contents = f.readlines()
    # Create adjacency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        right = right.strip()
        # Append inEdge
        graph.setdefault(right,[ [], 0 ])[0].append(left)
        # Increase outEdge
        graph.setdefault(left,[ [], 0 ])[1] += 1
    f.close()
    del line, contents
    del left, right

    # Graph size
    graph_size = len(graph)
    # Empty graph
    if graph_size == 0:
        return []

    # Collect dangling nodes, but don't add real edge in the graph.
    # Instead, calculate dangling_sum below.
    dangling_nodes = [node for node in graph.keys()\
        if graph[node][1] == 0]

    # Dict from nodename(key) to list index, avoid copying dicts.
    # e.g. :
    # graph_dict = { 'a':... , 'b':..., 'c':... }
```

```
    # weight_list = [ 0.33, 0.33, 0.33 ]
    # Only iterate weight_list.
    node_to_index = dict(zip(graph.keys(), range(graph_size)))

    # Initial weight
    old_weight = [1.0 / graph_size for i in range(graph_size)]

    # Tolerence of errorness.
    tol = 1e-6
    i = 0
    # Iterate
    while True:
        # Dangling sum from dangling nodes to every node.
        dangling_sum = 0
        for node in dangling_nodes:
            dangling_sum += old_weight[node_to_index[node]]
        # Base : (1-alpha) / n
        new_weight = [\
        (1.0 - damping_factor) / graph_size \
        for i in range(graph_size)]

        # Dangling nodes contribute to other nodes.
        # alpha * dangling_sum / graph_size
        dangling_avg = damping_factor / graph_size * dangling_sum;
        # Iterate nodes.
        for node in graph.keys():
            # Iterate innodes.
            for innode in graph[node][0]:
                # Add weights.
                new_weight[node_to_index[node]] += \
                    damping_factor / graph[innode][1] \
                    * old_weight[node_to_index[innode]]
            new_weight[node_to_index[node]] += dangling_avg

        delta = sum([abs(new_weight[i] - old_weight[i])\
            for i in range(graph_size)])
        if delta < tol * graph_size:
```

```
            break
        # Update.
        old_weight = new_weight
        i += 1
    del tol, i

    # Sort.
    sorted_result = \
        sorted( dict(zip(graph.keys(), new_weight)).items(), \
        key = lambda i: (i[1], i[0]), \
        reverse = True )
    # Generate list and score.
    # Comment 'scores' in homework code.
    node_list_in_descending_order = [term[0] for term in sorted_result]
    scores = [term[1] for term in sorted_result]
    return node_list_in_descending_order , scores


#@param input_Graph: 描述一个graph的纯文本邻接表文件名,如’E:\graph.txt’。
#@param input_Seed: 描述一个种子集的纯文本文件名,如’E:\seed.txt’。
#@param damping_factor
def PPR( input_Graph , input_Seed , damping_factor ) :
    # Read graph like PageRank(...).
    # graph = {'node': [[inEdge], outDegree]}
    graph = {}

    # Open and read file.
    with open(input_Graph, 'r') as f:
        contents = f.readlines()
    # Create adjacency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        right = right.strip()
        # Append inEdge
        graph.setdefault(right,[ [], 0 ])[0].append(left)
```

```
        # Increase outEdge
        graph.setdefault(left,[ [], 0 ])[1] += 1
    f.close()

    # Graph size
    graph_size = len(graph)
    # Empty graph
    if graph_size == 0:
        return []

    # Read seed.
    # seed = {'node' : weight}
    seed = {}
    with open(input_Seed, 'r') as f:
        contents = f.readlines()
    # Create adjacency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        weight = float(right)
        # Add weight.
        seed[left] = weight
    f.close()
    del line, contents
    del left, right, weight

    # Collect dangling nodes, but don't add real edge in the graph.
    # Instead, calculate dangling_sum below.
    dangling_nodes = [node for node in graph.keys()\
        if graph[node][1] == 0]

    # Dict from nodename(key) to list index, avoid copying dicts.
    # e.g. :
    # graph_dict = { 'a':... , 'b':..., 'c':... }
    # weight_list = [ 0.33, 0.33, 0.33 ]
    # Only iterate weight_list.
```

```
    node_to_index = dict(zip(graph.keys(), range(graph_size)))

    # Initial weight using seed
    p0 = [0 for _ in range(graph_size)]
    for node in seed.keys():
        p0[node_to_index[node]] = seed[node]
    old_weight = p0

    # Tolerence of errorness.
    tol = 1e-6
    i = 0
    # Iterate
    while True:
        # Dangling sum from dangling nodes to every node.
        dangling_sum = 0
        for node in dangling_nodes:
            dangling_sum += old_weight[node_to_index[node]]
        # Base : (1-alpha) * p0
        new_weight = [\
        (1.0 - damping_factor) * p0[i]\
        for i in range(graph_size)]

        # Dangling nodes contribute to other nodes.
        # alpha * dangling_sum / graph_size
        dangling_avg = damping_factor / graph_size * dangling_sum;
        # Iterate nodes.
        for node in graph.keys():
            # Iterate innodes.
            for innode in graph[node][0]:
                # Add weights.
                new_weight[node_to_index[node]] += \
                    damping_factor / graph[innode][1] \
                    * old_weight[node_to_index[innode]]
            new_weight[node_to_index[node]] += dangling_avg

        delta = sum([abs(new_weight[i] - old_weight[i])\
            for i in range(graph_size)])
```

```python
        if delta < tol * graph_size:
            break
        # Update.
        old_weight = new_weight
        i += 1
    del tol, i

    # Sort.
    sorted_result = \
        sorted( dict(zip(graph.keys(), new_weight)).items(), \
        key = lambda i: (i[1], i[0]), \
        reverse = True )
    # Generate list and score.
    # Comment 'scores' in homework code.
    node_list_in_descending_order = [term[0] for term in sorted_result]
    scores = [term[1] for term in sorted_result]
    return node_list_in_descending_order , scores

# Test PR.
import time
time_start = time.time()
result = PageRank('soc-Epinions1_processed.txt', 0.85)
result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
time_end = time.time()
print( 'Time used for PageRank: %.3f s.' %(time_end - time_start) )
print('Node\t\tScore')
for k, v in result_dict_rank10.items():
    print('%s\t\t%.8f' % (k, v))

print()

# Test PPR.
time_start = time.time()
result = PPR('soc-Epinions1_processed.txt', 'soc-Epinions1_seed.txt', 0.85)
result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
time_end = time.time()
print( 'Time used for PPR: %.3f s.' %(time_end - time_start) )
```

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

2. 数据预处理——preprocess.py:

```python
# Preprocess data.

# Prepare graph
def prepare_graph():
    f_ori = open('soc-Epinions1.txt', 'r')
    contents = f_ori.readlines()
    f_ori.close()

    contents = contents[4:]
    contents = [line.replace('\t', ',') for line in contents]
    f_pre = open('soc-Epinions1_processed.txt', 'w')

    for line in contents:
        f_pre.write(line)
    f_pre.close()

# Generate seed.
def generate_seed():
    f_seed = open('soc-Epinions1_seed.txt', 'w')
    for i in range(50):
        f_seed.write(str(i) + ',0.02\n')
    f_seed.close()

prepare_graph()
generate_seed()
```

3. 与 nx 库对比——nx_pr_compare.py:

```python
import networkx as nx

def nx_PR_benchmark(input_file_name , damping_factor):
    G = nx.DiGraph()
    edges = []
```

```
    # Open and read file.
    with open(input_file_name, 'r') as f:
        contents = f.readlines()
    # Create adjency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        right = right.strip()
        # Append inEdge
        edges.append((left, right))
    f.close()
    G.add_edges_from(edges)

    sorted_result = sorted(\
        nx.pagerank(G, alpha = damping_factor).items(), \
        key = lambda x: (x[1], x[0]), reverse = True)

    # Generate list and score.
    # Comment 'scores' in homework code.
    node_list_in_descending_order = [term[0] for term in sorted_result]
    scores = [term[1] for term in sorted_result]
    return node_list_in_descending_order , scores

def nx_PPR_benchmark(input_Graph, input_Seed, damping_factor):
    G = nx.DiGraph()
    nodes = {}
    edges = []
    # Open and read file.
    with open(input_Graph, 'r') as f:
        contents = f.readlines()
    # Create adjency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        right = right.strip()
```

```
        # Append inEdge
        edges.append((left, right))
        # Add node name.
        nodes[left] = 0
        nodes[right] = 0
    f.close()
    G.add_edges_from(edges)

    # Read seed.
    # seed = {'node' : weight}
    seed = {}
    with open(input_Seed, 'r') as f:
        contents = f.readlines()
    # Create adjacency list of a graph.
    for line in contents:
        # if line.strip() == '':
            #continue
        left, right = line.split(',')
        weight = float(right)
        # Add weight.
        seed[left] = weight
    f.close()

    nstart = nodes
    for node in seed:
        nstart[node] = seed[node]

    dangling = dict.fromkeys(nodes, 1 / len(nodes))
    sorted_result = sorted(\
        nx.pagerank(G, alpha = damping_factor, \
            personalization = seed, nstart = nstart, \
            dangling = dangling).items(), \
            key = lambda x: (x[1], x[0]), reverse = True)

    # Generate list and score.
    # Comment 'scores' in homework code.
    node_list_in_descending_order = [term[0] for term in sorted_result]
```

```
    scores = [term[1] for term in sorted_result]
    return node_list_in_descending_order , scores

    return None


# Test nx PR.
import time
time_start = time.time()
result = nx_PR_benchmark('soc-Epinions1_processed.txt', 0.85)
result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
time_end = time.time()
print( 'Time used for nx PageRank: %.3f s.' %(time_end - time_start) )
print('Node\t\tScore')
for k, v in result_dict_rank10.items():
    print('%s\t\t%.8f' % (k, v))

print()

# Test nx PPR.
time_start = time.time()
result = nx_PPR_benchmark('soc-Epinions1_processed.txt', 'soc-Epinions1_seed.
    txt', 0.85)
result_dict_rank10 = dict(zip(result[0][:10], result[1][:10]))
time_end = time.time()
print( 'Time used for nx PPR: %.3f s.' %(time_end - time_start) )
print('Node\t\tScore')
for k, v in result_dict_rank10.items():
    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.