|  |  |
| --- | --- |
|  |  |
| **PageRank**  **实验报告** |  |
|  |  |
|  | **2020.06.22**  **数据科学导论** |
|  | **思子华**  **2018202181** |

目录

[1 实验要求 3](#_Toc43660330)

[1.1 作业内容 3](#_Toc43660331)

[2 实现思路 4](#_Toc43660332)

[2.1 PageRank 4](#_Toc43660333)

[2.1.1 核心功能 4](#_Toc43660334)

[2.1.2 代码结构 4](#_Toc43660335)

[2.1.3 代码实现思路 5](#_Toc43660336)

[为什么需要考虑dead ends 5](#_Toc43660337)

[为什么使用哈希表存储临接链表 6](#_Toc43660338)

[2.1.4 核心代码展示 7](#_Toc43660339)

[2.1.5 结果展示 8](#_Toc43660340)

[2.1.6 如何应对大规模图数据的设想和实现 8](#_Toc43660341)

[2.2 Personalized Page Rank 10](#_Toc43660342)

[2.2.1 核心功能 10](#_Toc43660343)

[2.2.2 运行结果 10](#_Toc43660344)

[3 PPR线性可加证明题 11](#_Toc43660345)

[4 实验小结 12](#_Toc43660346)

# 实验要求

## 作业内容

* 实现PageRank
* 实现Personalized PageRank
* 证明Personalized PageRank的线性可加性
* 考虑如何存储稀疏图，并对PageRank算法计算的部分进行优化

# 实现思路

## PageRank

### 核心功能

* 读取文件
* 从图中提取出dead ends
* 进行PageRank
  + 初始化向量
  + 迭代计算
  + 判断是否converge
* 返回根据分数降序排序的list

### 代码结构

|--> read\_file

|--> deadEnds

PageRank ----

|--> mypagerank (iterations)

|--> sort (sorted by value)

### 代码实现思路

* 从输入文件中提取出临接链表（使用哈希表存储）、提取所有结点的名称（存储在哈希表中）
* 根据临接链表从所有结点中挑选出只有入度没有出度的deadEnds
* 迭代计算pagerank
  + 按的比例计算
  + 考虑到deadEnd的存在，将deadEnds中的值均匀加到所有结点中
  + 按的比例
* 计算误差是否达到10e-6

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 为什么需要考虑dead ends 课上的ppt中没有处理dead end，但是作业的图中有1w多个结点是dead ends。如果不考虑dead ends，会导致向量P的各个维度之和从1不断减小，因为dead ends中值没有进一步传播出去。  根据实际测试，如果不考虑dead ends不会影响最终的迭代结果，但是会导致迭代次数增加。因为不考虑dead ends的情况下每一次迭代总P的各个维度的值都小于等于考虑dead ends的情况下P中各个维度的值，进而导致两次迭代之间的差距较小，因此迭代速度较慢。  如Figure1所示，图上方不考虑dead ends的时候迭代了43次，用时6.4s，并且排序结果的得分较低；图下方考虑dead ends以后，迭代了35次，用时5.56s，排序结果与上方相同，但是得分较高。(注：这里的图是PPR的运行结果) | |  |

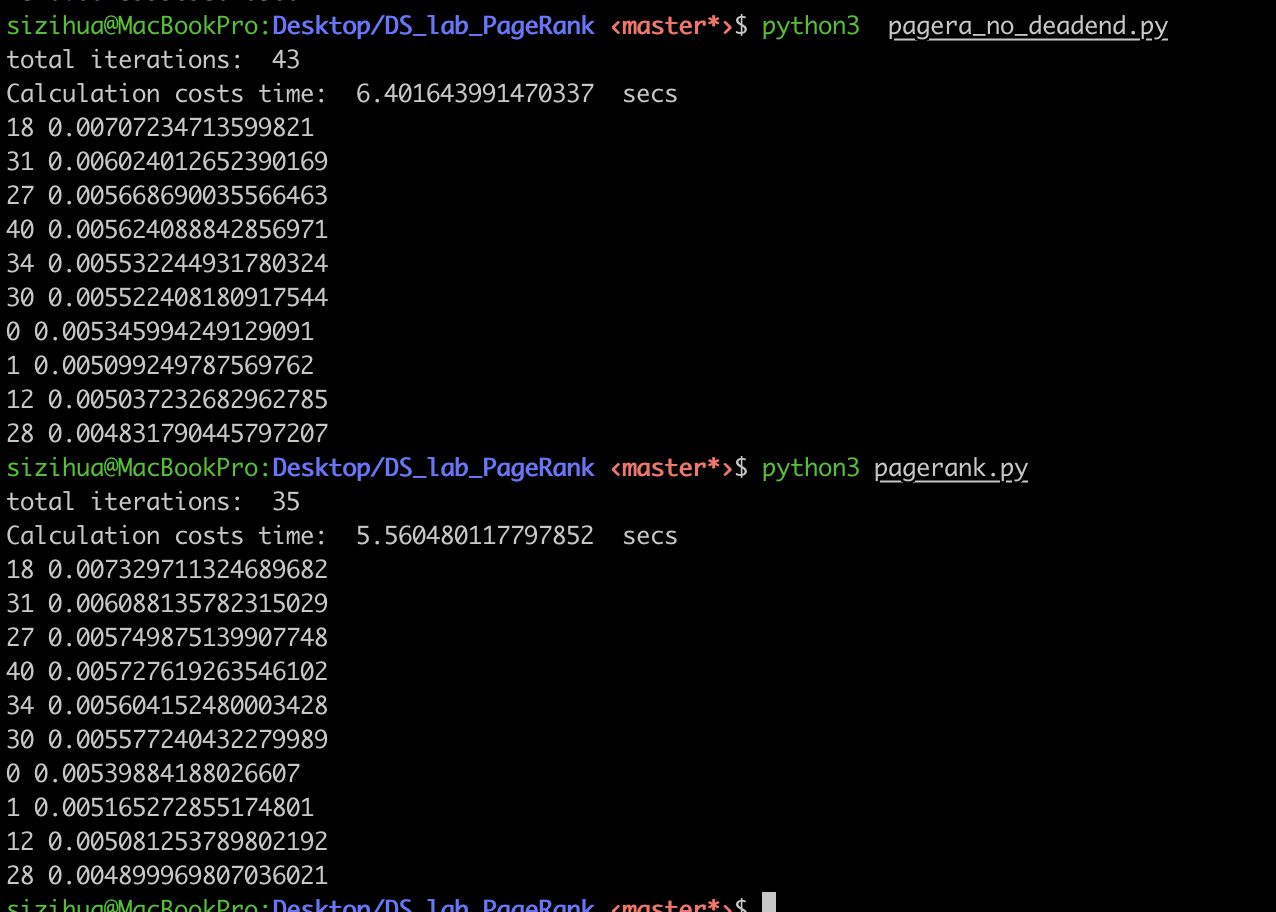


Figure 1

|  |  |
| --- | --- |
|  |  |
| 为什么使用哈希表存储临接链表 由于本次实验的数据为稀疏图（边数508837远小于结点数的平方），使用邻接链表存储图的数据。在迭代计算P与L的乘积时，从pagerank的实际含义出发，遍历临接链表，设遍历到结点时，将添加到中（其中，val表示结点到值，out\_degree表示出度，存在边（）。  程序中使用python自带的dict存储临接链表存储数据，由于dict是使用哈希表实现的，查询元素、修改value值的时间复杂度是O(1)。因此，本程序运行速度较快。 | |

### 核心代码展示

1. **def** mypagerank(link\_dict, pagelist, dead\_end\_list, beta=0.85):
2. #迭代计算
3. n\_pages = len(pagelist)  # 页面总数
5. r\_old = dict()
6. r\_new = dict()
7. **for** each\_page **in** pagelist.keys():
8. r\_old[each\_page] = 1/n\_pages
9. convergence = False
10. count = 0
11. **while** **not** convergence:
12. dead\_end\_sum = 0.0
13. **for** each\_end **in** dead\_end\_list:
14. dead\_end\_sum += beta \* r\_old[each\_end] / n\_pages
16. # 初始化 r\_new
17. **for** each\_page **in** pagelist.keys():
18. r\_new[each\_page] = (1 - beta) / n\_pages + dead\_end\_sum
19. **for** src **in** link\_dict.keys():
20. dest\_list = link\_dict[src]
21. src\_out\_degree = len(dest\_list)
22. **for** each\_dest **in** dest\_list:
23. r\_new[each\_dest] += beta \* r\_old[src] / src\_out\_degree
24. #判断是否converge
25. err = 0
26. threshold = 10e-6
27. **for** each\_page **in** pagelist.keys():
28. err += abs(r\_old[each\_page] - r\_new[each\_page])
29. **for** each\_page **in** pagelist.keys():
30. r\_old[each\_page] = r\_new[each\_page]
31. convergence = err < threshold
32. count += 1
33. #print('iteration: ', count, end='\r')
34. #print('total iterations: ', count)
35. **return** r\_new

时间复杂度分析

* 初始化时间复杂度O(V)
* 迭代循环中
* 计算dead ends中所有结点值的和需要O(V)
* 计算 并且加入dead ends中的值需要遍历一遍临接链表，时间复杂度O(V+E)
* 按的比例，时间复杂度为O(V)
* 由于一共迭代常数次，总的时间复杂度为O(V+E)

### 结果展示

* Page Rank运行结果



Figure 2

### 如何应对大规模图数据的设想和实现

一般情况下的PageRank时间、空间开销

* 1）时间开销: 算法主要时间开销将会是每轮迭代中做的矩阵乘法上, 这是O()的时间开销(V代表图中结点个数, 下同), 再乘上算法需要迭代k轮完成收敛, 因此PageRank的时间开销是 O(). 不过, 一般来说, 这个收敛需要次数k会是在10~100之间的数值, 不会特别大.
* 2)空间开销: 算法最大的空间开销来自于存储整个P矩阵到内存中, 这是的空间开销. 因此如果假设有10^6个结点的图, 需要的M矩阵大小是10^12, 按照int型4byte来存储, 这相当于4TB的内存开销, 这是无法承受的空间开销.

对矩阵空间优化的办法

* 已知大多数情况下, M矩阵十分稀疏, 那么我们可以使用邻接链表或者类似形式, 只存储非零元素的值.
* 比如, 在Python中, 可以通过构造字典数据类型来实现.
  + G = {1: [2, 3, 4], 2: [1, 4], 3: [1], 4: [2, 3]}  
    G[1] = [2, 3, 4]表示结点1和2, 3, 4是有一条有向边.
* 那么这种情形下, 空间开销就是O(V+E)的, 因为邻接链表存储了所有的点和有向边. 在稀疏图中, O(V+E)往往远小于O(V^2).

对时间的优化办法

* 为了实现从O(V^2)下降到O(V+E)的优化, 我们需要重新定义一般PageRank中的PL矩阵乘法操作.
* 我们知道, PL乘法实际上完成的目的是算出P\*列向量, 也就是每个结点新的PageRank值. 按照之前所述, 我们有如下观察:
* Observation : 结点的新PageRank值 = Σ (来源结点的PageRank值 • 本结点所分享到的权重)

1. **for** src **in** link\_dict.keys():
2. dest\_list = link\_dict[src]
3. src\_out\_degree = len(dest\_list)
4. **for** each\_dest **in** dest\_list:
5. r\_new[each\_dest] += beta \* r\_old[src] / src\_out\_degree

* 由此, 我们的时间开销变成了O(V + E), 在稀疏图中, 这样的时间开销远比O(V^2)小

设想

本次实验的数据规模并没有特别的大，如果真的遇到特别大的数据规模，可以考虑使用并行、分布式等方式进行计算，并且可以使用特定的方式存储稀疏矩阵（比如三元组存储，十字链表存储法）

## Personalized Page Rank

### 核心功能

* 与PageRank基本一致
* 核心代码只有一处不同

变为

*那一部分代码更改如下*

1. **for** item in input\_seed.keys():
2. r\_init[item] = (1 - beta) \* input\_seed[item]

### 运行结果

* PPR



Figure 3

# PPR线性可加证明题

PPR公式如下：

不妨设 , 其中，。

下面用**数学归纳法**证明 ：

(其中k)

设 , 。

显然，m = 1时，满足。

归纳假设：若 m = k时，有成立。

则有：

那么m = k + 1时：

上式

根据（1）式，

上式

即成立，由数学归纳法可知得证。

综上所述，成立

# 实验小结

**本次实验，我认为有三个trick的点**

* 使用python自带的dict（哈希表）存储临接链表，使得随机访问元素和修改value的代价都是O(1)
* 迭代计算矩阵乘法时，结合page rank的原理进行了优化，使得矩阵乘法的时间复杂度降为了O(E+V)
* 计算时考虑到dead ends的影响，减少了迭代次数，增加了运行速度