南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

大数据计算和应用实验报告

PageRank 大作业

姓名：张丛

学号：2113662

专业：信息安全

指导教师：杨征路

2024 年 4 月 25 日

摘要

关键字：PageRank，Block-Stripe

目录

一、 实验内容

(一) 实验目的

在此次项目中，需要报告 PageRank 分数最高的前 100 个 NodeID。可以选择不同的参数（比如 TELEPORT），来比较不同的结果。

必须报告的一个结果是将 TELEPORT 设置为 0.85 时的结果。

除了基本的 PageRank 算法，还需要实现 Block-Stripe 算法。

(二) 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]

二、 PageRank 概述

(一) 什么是 PageRank

PageRank 是由谷歌公司创始人之一 Larry Page 提出的一种网页排序算法，用于衡量网页在搜索引擎结果中的重要性。

PageRank 基于网页之间的链接关系，通过分析网页被其他网页链接的数量和质量来评估网页的权重，从而在搜索引擎结果中为用户提供更相关和有质量的搜索结果。

PageRank 的核心思想是，一个被更多其他网页链接的网页通常比被较少链接的网页更重要。而这个权重的传递是通过迭代计算得到的。

(二) PageRank 算法描述

具体来说，PageRank 通过以下步骤进行计算：

1. 初始化阶段

给定一个包含 N 个网页的集合，初始化每个网页的 PageRank 值为 1/N。

2. 迭代计算

重复进行以下步骤直到收敛（即 PageRank 值不再发生显著变化）：

a. 对于每个网页 i，计算其新的 PageRank 值为：

$$PR(i) = \frac{(1-d)}{N} + d\sum\left(\frac{PR(j)}{L(j)}\right)$$

其中 j 为所有指向网页 i 的网页，L(j) 为网页 j 的出链数量，d 为阻尼因子（一般取值为 0.85）。

b. 更新每个网页的 PageRank 值为新计算的值。

3. 收敛阶段

当所有网页的 PageRank 值收敛到一个稳定状态时，算法停止。

4. 特殊处理

PageRank 算法的关键在于处理网页之间的循环链接和悬挂节点（即没有出链的网页），通常通过引入阻尼因子和均匀分配悬挂节点的 PageRank 值来解决这些问题。

(三) Block-stripe 算法

PageRank 的 Block-stripe 算法是一种优化方法，旨在加快 PageRank 算法的计算速度。

在传统的 PageRank 计算中，需要对整个图进行迭代计算，这在大规模图上会导致计算成本很高。Block-stripe 算法通过将图分割成多个块（blocks），然后在每个块上进行迭代计算，最后再将结果合并，从而降低计算复杂度。

具体来说，Block-stripe 算法的步骤如下：

1. 将整个图分割成多个块，每个块包含一部分节点和相应的边。
2. 在每个块上独立地进行 PageRank 计算的迭代过程，更新每个节点的 PageRank 值。
3. 将每个块的 PageRank 值进行合并，以获取整个图的最终 PageRank 值。

三、 数据集说明

数据集 data.txt 中的每行的数据格式为 <FromNodeID, ToNodeID>, 即从结点 FromNode 到结点 ToNode 的一条边。

数据集中含有重复数据，在实验中我们会对数据进行类似去重的操作，即视为同一条边。

对于悬挂结点（没有出度）和循环链接（结点环路），会引入随机跳转机制。

四、 实验过程

(一) 基本 PageRank 算法的实现

1. 主函数

基本上可以分为三个大流程：

1. 数据处理与加载
2. PageRank 算法实现
3. 保存结果

```
if __name__ == '__main__':

    if setup() == 0:
        # 加载图数据并计时
        graph = time_test("read_graph", load_graph)
        # 计算PageRank值并计时
        result = time_test("pagerank", pagerank, graph)
        # 按PageRank值进行排序
        result = sorted(result.items(), key=lambda x: x[1], reverse=True)
```

```
10
11          # 要输出的前n个PageRank值
12          topn = 100
13          if topn > 0:
14              result = result[:topn]
15
16          # 写出结果到文件
17          with open(BASIC_OUT, 'w', encoding='utf-8') as f:
18              for line in result:
19                  f.write(f"[{line[0]}] [{line[1]}]\n")
```

2. 数据加载

逻辑如下:

1. 打开数据集文件，读取每一行数据，将其解析为源节点和目标节点，并添加到 links 列表中。

2. 遍历 links 列表，获取其中唯一的节点列表 nodes ，用于初始化 Graph 对象。

3. 打印总链接数和唯一链接数。

4. 找到最大节点编号，作为节点数量，用于初始化 Graph 对象。

5. 返回一个 Graph 对象，表示从数据集中读取的图。

```
1   def load_graph() -> Graph:
2
3       # 从数据集中读取数据，并将其解析为边的列表
4       links = []
5       with open(DATA_IN, "r", encoding="utf-8") as file:
6           for line in file:
7               src, dst = map(int, line.split())
8               links.append((src, dst))
9
10          # 获取唯一的节点列表
11          nodes=[]
12          for row in links:
13              if row not in nodes:
14                  nodes.append(row)
15          print("num of total links: {}".format(len(links)))
16          print("num of unique links: {}".format(len(nodes)))
17
18          # 找到最大的节点编号
19          max_both = -1
20          for i in links:
21              max_both = max(max_both, i[0], i[1])
22
23          print("max node number: {}".format(max_both))
24          # 创建并返回Graph对象
25          return Graph(max_both, nodes)
```

3. 迭代计算 PageRank

主要流程如下：

1. 预处理：首先对图进行预处理，包括计算图中的唯一节点数量和初始化 PageRank 矩阵。

2. 迭代计算：进入迭代计算阶段，在每次迭代中，按照 PageRank 公式计算每个节点的新 PageRank 值，并进行调整以确保 PageRank 值的总和为 1。

3. 停止条件：在每次迭代后，计算当前迭代的最大误差，如果最大误差小于预设的阈值（EPSILON）或者达到最大迭代次数（MAX_ITER），则停止迭代。

4. 输出结果：将计算得到的 PageRank 值存储在字典 'result'中，并返回该字典作为结果。

```
def pagerank(graph: Graph):

    # 预处理
    i = 0
    count = 0
    is_isolate = np.full(graph.nnodes+1, True)
    while i < (graph.nnodes+1):
        if not (graph.out_degrees[i] == 0 and \
                len(graph.in_edges[i])==0): # 入度或出度不为0
            count += 1
            is_isolate[i] = False
        i += 1
    N = count
    print("number of unique nodes:{} ".format(count))

    # 初始化矩阵
    r_old = np.full(graph.nnodes+1, np.float64(1 / N))
    r_old[is_isolate] = 0

    iter = 0
    while True:
        r_new = np.zeros(graph.nnodes + 1, dtype=np.float64)
        for dst in range(graph.nnodes+ 1):
            if len(graph.in_edges[dst]) == 0:  # 入度为0
                r_new[dst] = 0
            else:
                for src in graph.in_edges[dst]:
                    # 计算新的PageRank值
                    r_new[dst] += (r_old[src] \
                        / graph.out_degrees[src])*TELEPORT
        s = np.float64(np.sum(r_new))
        print("Iter: {}, Before adjusting, S value: {:.17f}".format(iter, s))
        # 调整PageRank值，重新插入泄漏的PageRank值
        r_new += (1 - s) / N
        r_new[is_isolate] = 0
        s = np.float64(np.sum(r_new))
        print("Iter: {}, After adjusting, S value: {:.17f}".format(iter, s))

        iter += 1
```

```
40
41          # 计算最大误差
42          error = 0.
43          for i in range(graph.nnodes + 1):
44              if(error < abs(r_new[i] - r_old[i])):
45                  error = abs(r_new[i] - r_old[i])
46
47          if (error < EPSILON or iter >= MAX_ITER):
48              print(f"absolute error: {error}, iter: {iter}")
49              break
50          r_old = copy.deepcopy(r_new)
51
52      result = {}
53      for i in range(1, graph.nnodes+1):
54          result[i] = r_new[i]
55
56      return result
```

4. 输出 top100

即在主函数中对 result 按照 PageRank Score 进行排序，并保存前 100 名到指定文件。

(二) Block-Stripe 算法的实现

1. 数据预处理

在 Block-Stripe 算法中，对结点进行了映射和编号重新排列的操作，这个操作是为了是为了简化计算过程和降低复杂度。

可以体现在以下几个方面：

1. 数据结构：使用二维数组或稀疏矩阵表示节点之间的边关系。

2. 计算优化：重新编号，节点之间的关系在数组或矩阵中更容易映射到不同的处理单元或线程中进行计算，从而提高计算效率。

3. 减少处理复杂度：重新编号后，遍历节点时不需要考虑原始节点编号的顺序和连续性，简化了计算过程中的逻辑处理，减少了错误和调试的难度。

```
1   def preprocessing(data_txt, new_data_txt):
2       # 读取文件并建立节点编号映射关系
3       edges = []
4       with open(data_txt, 'r') as f:
5           for line in f:
6               # 读取每一行数据，并将起始节点和目标节点提取出来
7               src, dest = map(int, line.strip().split())
8               # 将边的起始节点和目标节点存储为元组，并添加到边列表中
9               if (src, dest) not in edges:
10                  edges.append((src, dest))
11
12      nodes = set()
13      for edge in edges:
```

```
        # 将边列表中的节点添加到节点集合中
        nodes.add(edge[0])
        nodes.add(edge[1])

    # 得到排序后的节点列表
    node_list = sorted(list(nodes))

    # 建立节点映射字典，将旧id映射为新id
    node_dict = {}
    for i, node_id in enumerate(node_list):
        node_dict[node_id] = i + 1

    # 将边列表中的节点id映射为新id
    new_edges = []
    for edge in edges:
        new_src = node_dict[edge[0]]
        new_dest = node_dict[edge[1]]
        new_edges.append((new_src, new_dest))

    # 将新的边列表写入文件
    with open(new_data_txt, 'w') as f:
        for edge in new_edges:
            new_src = edge[0]
            new_dest = edge[1]
            f.write(str(new_src) + " " + str(new_dest) + "\n")

    # 返回节点映射字典
    return node_dict
```

2. 数据分块

将处理过后的数据进行分块，如果当前块的目标节点数达到 block_size，则将当前块的数据保存到磁盘中，并开始处理下一个块。

同时生成用于后续计算的数据结构，同时将一些中间结果保存到文件中，以便后续使用。

```
def pre_block(data):

    pages = -1  # 初始化最大节点编号为-1
    name_count = 0  # 初始化块文件名计数器为0
    s_matrix = defaultdict(list)  # 使用defaultdict创建空的字典，用于存储块数据
    out_of_nodes = defaultdict(int)  # 使用defaultdict创建空的字典，用于记录节点的出度
    files_of_des = defaultdict(list)  # 使用defaultdict创建空的字典，用于记录每个节点在哪些块文件中出现
    current_num_des_nodes = []  # 初始化当前块中的目标节点列表

    with open(data, 'r', encoding='utf-8') as f:
```

```
        for line in f:
            # 读取每一行数据，并提取出起始节点和目标节点
            src = int(line.split()[0].split()[0], 10)
            des = int(line.split()[1].split()[0], 10)
            # 更新最大节点编号
            pages = max(src, pages, des)
            # 记录节点的出度
            out_of_nodes[src] += 1
            # 记录每个目标节点在哪些块文件中出现
            if name_count not in files_of_des[des]:
                files_of_des[des].append(name_count)

            # 将数据按照规则分块处理
            if des not in current_num_des_nodes:
                if len(current_num_des_nodes) % block_size == 0:
                    if len(current_num_des_nodes) != 0:
                        # 如果当前块的目标节点数达到规定的块大小，保存当前块的数据到磁盘中
                        save_data_for_RI(s_matrix, name_count)
                        name_count += 1
                        # 清空内存中的数据，为处理下一个块做准备
                        s_matrix = defaultdict(list)
                        current_num_des_nodes = []
                        # 将当前数据加入到新的块中
                        s_matrix[des].append(src)
                        current_num_des_nodes.append(des)
                    else:
                        # 如果当前块中没有数据，直接将数据加入到内存中
                        s_matrix[des].append(src)
                        current_num_des_nodes.append(des)
                else:
                    # 当前块的目标节点数未达到规定的块大小，将数据加入到内存中
                    s_matrix[des].append(src)
                    current_num_des_nodes.append(des)
            else:
                # 当前目标节点已在块中，直接将数据加入到内存中
                s_matrix[des].append(src)

        # 处理最后一个块
        save_data_for_RI(s_matrix, name_count)

        # 将最大节点编号保存到文件中
        with open(pages_txt, 'w', encoding='utf-8') as pages_file:
            pages_file.write(str(pages))

        # 清空内存中的数据
        s_matrix = defaultdict(list)
```

```
        current_num_des_nodes = []

    # 将数据保存到不同的文件中
    process_block_data(pages, compute_size, files_of_des, out_of_nodes)
```

3. 迭代计算 PageRank

主要流程:

1. 计算块数量，初始化结点 PageRank 值
2. 迭代计算：重复以下步骤直到收敛为止:
   - 对于每个节点 ‘i‘，计算其新的 PageRank 值 ‘PR(i)‘:

$$\mathrm{PR}(i) = \frac{1-d}{N} + d \times \sum_{j \in M(i)} \frac{\mathrm{PR}(j)}{L(j)}$$

   - 更新所有节点的 PageRank 值。
   - 计算误差：计算本次迭代的误差 ‘e‘，并根据误差判断是否继续迭代。
   - 如果误差小于预先设定的阈值 ‘E‘，迭代停止；否则，继续下一轮迭代。
3. 保存计算结果

```
def sparse_matrix_multiply(size, N, M):
    ……

    # 初始化 r_old, 将每个节点的PageRank值初始化为1/N
    temp = np.full(N, 1/N, dtype=np.float64)
    with open('./r_old.txt', 'w', encoding='utf-8') as t:
        for i in temp:
            t.write(str(i) + '\n')

    # 计算需要处理的块的数量 K
    K = N // size
    if N % size != 0:
        K += 1

    E = 0.0001  # 定义迭代终止条件中的误差阈值

    # 迭代计算
    if_end = 0
    while not if_end:
        r_new = np.zeros(size)  # 初始化本次迭代的 r_new
        start = 1  # 每一块 r_new 的起始索引
        end = start + size - 1  # 每一块 r_new 的结束索引
        S = 0  # 初始化本次迭代的总PageRank值

        for matrixs in range(len(os.listdir(M))):
            # 遍历每个矩阵文件
            m_now = M + str(matrixs) + '.txt'  # 获取当前矩阵文件的路径
            with open(m_now, 'r', encoding='utf-8') as f1:
                matrix = json.loads(f1.read())  # 读取矩阵数据
```

```
30
31              # 处理当前矩阵文件中的每一行数据
32              if matrix == {}:
33                  continue
34              else:
35                  with open('./r_old.txt', 'r', encoding='utf-8') as f2:
36                      count = 1
37                      i = f2.readline().rstrip()
38                      for src in matrix:
39                          while int(src, 10) != count:
40                              count += 1
41                              i = f2.readline().rstrip()
42                          if int(src, 10) == count:
43                              d = matrix[src][0]  # 获取源节点的出度
44                              i = np.float64(i)
45                              for des in matrix[src][1]:
46                                  t = des
47                                  if t > end:
48                                      # 如果目标节点超出当前块的范围, 重新定位
                                            块范围
49                                      c = t - start
50                                      c = c // size * size
51                                      with open('./r_new_temp.txt', 'a',
                                            encoding='utf-8') as f3:
52                                          for x in range(c):
53                                              f3.write('0.0\n')
54                                      start += c
55                                      end = start + size - 1
56                                  r_new[des - start] += b * i / d  # 更新目标节
                                        点的 PageRank 值
57                              count += 1
58                              i = f2.readline().rstrip()
59                          else:
60                              count += 1
61
62              start += size
63              end = start + size - 1
64
65              if end > N:
66                  end = N
67
68              # 将当前块计算得到的 PageRank 值写入临时文件, 并累加到总 PageRank
                    值 S 中
69              with open('./r_new_temp.txt', 'a', encoding='utf-8') as f3:
70                  for j in r_new:
71                      f3.write(str(j) + '\n')
72                      S += j
73
```

```
            # 如果当前块超出总页数 N, 则跳出循环
            if start <= end:
                r_new = np.zeros(end - start + 1)

        # 计算当前迭代的误差 e
        e = 0.0
        with open('./r_new_temp.txt', 'r', encoding='utf-8') as t:
            with open('./r_new.txt', 'w', encoding='utf-8') as new:
                for i in t:
                    i = np.float64(i)
                    new.write(str((i + (1 - S) / N)) + '\n')

        # 更新 r_old
        with open('./r_new.txt', 'r', encoding='utf-8') as new:
            with open('./r_old.txt', 'r', encoding='utf-8') as old:
                for i, j in zip(new, old):
                    tmp = abs(np.float64(i)-np.float64(j))
                    if e < tmp:
                        e = tmp

        with open('./r_new.txt', 'r', encoding='utf-8') as new:
            with open('./r_old.txt', 'w', encoding='utf-8') as old:
                old.truncate()
                for i in new:
                    i = np.float64(i)
                    old.write(str(i) + '\n')

        # 如果误差小于阈值 E, 则终止迭代
        if abs(e) <= E:
            if_end = 1
        else:
            with open('./r_new.txt', 'w', encoding='utf-8') as new:
                new.truncate()
            with open('./r_new_temp.txt', 'w', encoding='utf-8') as new:
                new.truncate()

    return r_new, S
```

4. 输出 top100

将得到的结果按照 PageRank Score 进行排序，再将 nodeID 映射回原来的 nodeID，输出结果，保存文件。

```
    result = dict()
    result = dict(sorted(r.items(), key=lambda d: d[1], reverse=True))  # 按照rank排序

    # 映射回原来的 nodeID
```

```
    reverse_mapping = {v: k for k, v in node_dict.items()}
    result_original = {}
    for mapped_node, pagerank in result.items():
        original_node = reverse_mapping[mapped_node]
        result_original[original_node] = pagerank

    with open(top100_result_file, 'w', encoding='utf-8') as f:
        count = 0
        for i in result_original:
            if count < 100:
                f.write('[' + str(i) + '] [' + str(result_original[i]) + ']\n
                    ')
                count += 1
            else:
                break

    # 保存文件
    with open(result_data_file, 'wb') as f:
        pickle.dump(result_original, f)
```

五、 实验结果及分析

(一) 基本 PageRank 实验结果

如下:

排名	结点序号	PageRank 值
1	2730	0.0004913431709331208
2	7102	0.0004784238756203468
3	1010	0.0004756056526635286
4	368	0.0004687634847338629
5	7453	0.00046875092835052925
6	1907	0.0004647386280189528
7	4583	0.00046183508717343716
8	1847	0.00046172121814305663
9	5369	0.00045852963552418476
10	3947	0.00045575081702916865

表 1: PageRank 排名

(二) Block-Stripe 实验结果

如下:

排名	结点序号	PageRank 值
1	2730	0.0004915762331518031
2	7102	0.00047837670774598034
3	1010	0.0004753914071688646
4	368	0.00046898779353740714
5	7453	0.0004689532213259118
6	1907	0.00046495839448257726
7	1847	0.00046193403934502707
8	4583	0.0004614396479094276
9	5369	0.00045876761944444464
10	3947	0.0004554186668593386

表 2: PageRank 排名

(三) 分析

1. 对比两组 PageRank 值，可以看到在两组结果中，页面的整体排名趋势是一致的（比如 top10 和 top100），即在两组结果中排名靠前的页面通常是重要页面，即便算法有些许不同但重要页面也应该靠前。

2. 尽管整体趋势一致，但两组结果（top100）中的 PageRank 排名还是有所不同的。这可能是由于 Block-Stripe 算法在处理大规模数据时对内存和计算资源的更有效利用，以及对稀疏矩阵的更好处理，从而使得部分页面的 PageRank 值略有变化。

尽管有细微差异，但两种算法得出的结果应该是相对可靠的。PageRank 算法本身具有一定的随机性，而且对于大规模数据，不同的实现可能会产生一些差异。

3. 算法效率：Block-Stripe 算法相对于基本的 PageRank 程序可能具有更高的计算效率和扩展性，特别是在处理大规模数据时。它通过分块处理数据，减少了内存和计算资源的消耗，提高了计算速度。

4. 在实验中，使用的 TELEPORT 值为 0.85。在实际应用中，应该根据具体需求选择合适的算法和参数来得出最符合实际情况的结果。

六、 总结

本次实验实现了 PageRank 算法，并通过调整参数（如 TELEPORT）来比较不同的结果。除了基本的 PageRank 算法，还实现了 Block-Stripe 算法。在实验中考虑了稀疏矩阵的优化存储和计算、处理 dead ends 和 spider trap 节点、实现分块计算等方面。

在实验中，首先实现了基本的 PageRank 算法，并将 TELEPORT 参数设置为 0.85，得到了 PageRank 分数最高的前 100 个 NodeID。随后，实现了 Block-Stripe 算法，并比较了两种算法的结果。

结果显示，两种算法得到的结果整体趋势一致，但存在细微差异，这可能是由于 Block-Stripe 算法在处理大规模数据时对内存和计算资源的更有效利用导致的。

通过本次实验，我深入理解了 PageRank 算法的原理和实现细节，掌握了稀疏矩阵的优化存储和计算方法，以及分块计算的实现方式。同时，也了解到了在实际应用中如何调整参数来影响 PageRank 算法的结果。