

Pagerank report

实验概要

本次实验研究Pagerank算法在web网页中权值排序中的应用，并且分别实现了Pagerank基本算法和block stripe分块算法。我们首先实现了Pagerank基本算法，然后结合实际场景中内存不足的情况，我们实现了block stripe分块的Pagerank算法。我们将两种算法的最终结果进行了比较，结合其运行速度等情况我们也思考了Pagerank算法的未来改进方向。

实验相关统计信息

数据集

数据格式

本次实验的数据集为 `data.txt`，其数据格式为每行按照<FromNodeID ToNodeID>的方式进行存储，其表示graph中存在从节点FromNodeID到ToNodeID的一条边。

节点个数及最大最小索引

经过我们的统计，本数据集中节点的最大索引为8297，最小索引为1，出现的节点总个数为8297。

spider trap以及dead end

本数据集中存在只有入度但没有出度的节点，即**dead end**；同时还存在没有出度的节点集，即**spider trap**。这两种情况不加处理都会造成不利影响。如果跳转到**dead end**不加处理，则之后无法再进行跳转；如果跳转到**spider trap**不加处理，则之后只能继续在该节点集内部进行跳转而不能跳转到其以外的节点。本次实验中，我们为了解决这两种情况，引入了一下方法：

1. 引入**beta**来模拟随机跳转因子，从而解决**spider trap**问题；
2. 对r进行归一化操作来解决**dead end**问题。

实验原理

Pagerank算法的总步骤其实可以分为以下几个步骤：

1. 初始化所有web网页的Pagerank值为 $1/N$ ，其中N为总网页数；
2. 根据网页之间的关系得出每个网页的出度，以及其指向的节点；
3. 设置收敛条件，迭代计算每个网页的Pagerank值，每次都进行归一化处理；
4. 计算所有网页的新旧Pagerank值的差值的绝对值的和，然后将其与收敛条件进行比较，如果满足收敛条件就结束迭代过程。

本次实验中我们对Pagerank算法进行了基本算法和block stripe算法进行了实现，同时还有对block stripe算法的加速实现。下面我们将分别对其进行介绍：

basic Pagerank

算法介绍

PageRank: The Complete Algorithm

□ Input: Graph G and parameter β

- Directed graph G (can have **spider traps** and **dead ends**)
- Parameter β

□ Output: PageRank vector r^{new}

- **Set:** $r_j^{old} = \frac{1}{N}$ 蜘蛛网和黑洞均能解决
- **repeat until convergence:** $\sum_j |r_j^{new} - r_j^{old}| > \varepsilon$
 - $\forall j: r_j^{new} = \sum_{i \rightarrow j} \beta \frac{r_i^{old}}{d_i}$
 $r_j^{new} = 0$ if in-degree of j is 0
 - **Now re-insert the leaked PageRank:**
 $\forall j: r_j^{new} = r_j^{new} + \frac{1-S}{N}$ **where:** $S = \sum_j r_j^{new}$
 - $r^{old} = r^{new}$

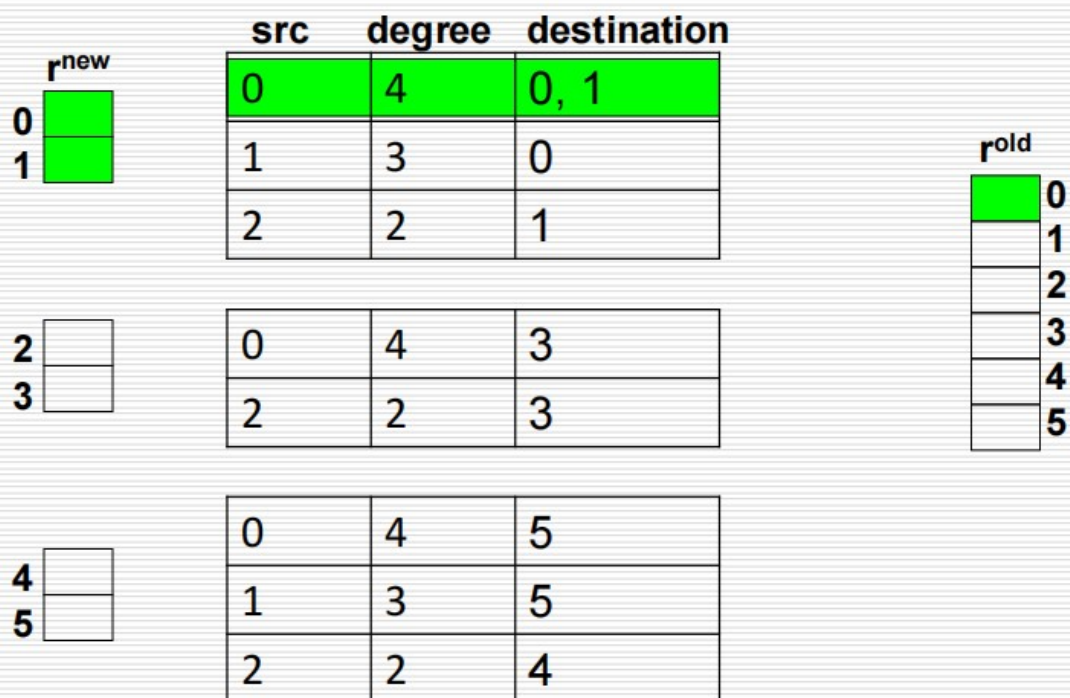
If the graph has no dead-ends then the amount of leaked PageRank is $1-\beta$. But since we have dead-ends the amount of leaked PageRank may be larger. We have to explicitly account for it by computing S .

54

我们实现的基本算法如上所示，此算法完美解决了spider traps问题和dead ends问题。基本逻辑是最开始对r进行初始化，然后迭代计算，首先计算beta*M之后的r的值，然后对此r进行归一化处理。具体是利用到S如上进行归一化，使所有节点的Pagerank值的和为1。每次计算完后，我们会计算是否满足收敛条件，并且对r进行更新。

block stripe pagerank

Block-Stripe Update Algorithm



Break M into stripes! Each stripe contains only destination nodes in the corresponding block of r^{new}

60

我们的分块算法的数据组织形式如上所示，我们依据对 r 的分组，也相应地对 M 进行分组。把与每组 r 更新相关的 M 组织成如上形式进行一一对应。当我们对某一组的 r 进行更新时，我们便可以定位与此相关的分组 M ，然后进行更新即可。

我们对spider traps问题和dead ends问题的解决方法与基本算法完全一致，block stripe算法就是基本算法在内存不足的情况下的一种变形计算方式。

block stripe pagerank加速算法

算法介绍

如果我们每次可以将 M 的一个分组读入内存，那么我们就可以使用这个算法来大大加快速度。我们对某一组 r_{new} 进行计算时，我们来循环确定 M 中的src是否属于某一相同的组，我们将其组成一个列表，然后导入某一块的 r 值进入内存。这样的话，对于某一组 r_{new} 的更新我们可以做的不重复的读取某一组的 r 。于是达到了加速的目的。其中我们主要的修改如下：

1. M 分块读取，不再按行读取；
2. 分开 r_{new} 计算时，如果src属于某一相同的块，直接导入此块的 r 值，然后一并更新。

关键部分代码解析

我们按照上述解释的那样，分basic Pagerank，block stripe pagerank以及block stripe pagerank加速算法来解释。

basic Pagerank

代码实现

我们Pagerank基本算法的实现是建立在内存充足的基础上的，即我们可以把所有节点的Pagerank值和他们的M矩阵都读入到内存。每次计算的时候其实便可以进行矩阵运算来达到加速运算的目的。

导入数据集

在读取data.txt的数据时，我们获取到其中的节点关系以及节点总数目。并且我们根据这些建立link_matrix矩阵。其每一个元素是一个列表，里面存储着对应节点的索引、出度还有到达的所有节点。其中到达的所有节点组成一个列表。具体代码如下：

```
1  def load_data():
2      # Load the data from the file
3      global max_node_index
4      global Num
5      global nodes_set
6      link_matrix = [[i,0,[]] for i in range(max_node_index+1)] #link matrix
7      src degree dest[]
8      max_node_index=0
9      with open(data_path) as f:
10         lines = f.readlines()
11         for line in lines:
12             split_line = line.split()
13             fr, to = int(split_line[0]), int(split_line[1])
14             link_matrix[fr][1]+=1 #degree
15             link_matrix[fr][2].append(to) #dest
16             max_node_index = max(max_node_index,fr,to)
17         Num=max_node_index
18         #把link_matrix中出度为0的节点删除
19         link_matrix=[link_matrix[i] for i in range(max_node_index+1) if
20 link_matrix[i][1]>0]
21         with open(LINK_MATRIX_PATH, "wb") as f:
22             for row in link_matrix:
23                 pickle.dump(row, f)
```

Pagerank值初始化

开始进行迭代前，我们首先将所有节点的Pagerank值都初始化为 $1/Num$ 。

```
1 def basic_pagerank():
2     #initialize the r_old
3     r_old = np.ones(Num+1)*1.0 / Num
4     r_old[0] = 0.0
5     ...
```

r_new计算

我们r_new的计算遵照上述算法的内容，首先依据每个节点的出度对其dest的节点的Pagerank值进行计算。然后再以此进行归一化操作得到新一轮的r的值。

首先依据每个节点的出度对其dest的节点的Pagerank值进行计算：

```
1 def compute_rnew(r_old):
2     r_new = np.zeros(Num+1)*1.0
3     with open(LINK_MATRIX_PATH, 'rb') as f:
4         while True:
5             try:
6                 # 逐行读取
7                 row_data = pickle.load(f)
8                 # print(row_data)
9                 src, degree, dest = row_data
10                for i in range(degree):
11                    r_new[dest[i]] += beta * r_old[src] / degree
12            except EOFError:
13                break # 如果到达文件末尾，跳出循环
14        ...
15    #re-insert the leaked
16    sum=np.sum(r_new)
17    sum-=r_new[0]
18    leaked=np.ones(Num+1)*(1-sum)/Num
19    r_new+=leaked
20    r_new[0] = 0.0
21    return r_new
```

归一化操作：

```

1      #re-insert the leaked
2      sum=np.sum(r_new)
3      sum-=r_new[0]
4      leaked=np.ones(Num+1)*(1-sum)/Num
5      r_new+=leaked
6      r_new[0] = 0.0
7      ...

```

综上，我们便完成了新一轮r的更新。

收敛条件判断

如下便是我们对收敛条件的判断：

```

1  while np.sum(np.abs(r_new - r_old)) > epsilon:
2      print(np.sum(np.abs(r_new - r_old)))
3      r_old = r_new
4      r_new = compute_rnew(r_old)

```

最终结果写入文件

本实验中我们对top100的节点的索引以及其Pagerank值写入文件，具体实现函数如下：

```

1  def print_result(r_new):
2      #降序排列
3      index = np.argsort(-r_new)
4      #output the result
5      for i in range(100):
6          print("The No.%d node is %d, and the pagerank is %.10f" % (i, index[i],
7              r_new[index[i]]))
8          with open(RESULT_PATH, "w") as f:
9              for i in range(100):
1             f.write(f"{index[i]} {r_new[index[i]]}\n")

```

如上，我们首先对r_new中的Pagerank值取相反数，然后调用np.argsort函数便可以实现降序排序，然后我们根据获取的index数组来将top100的节点的索引以及Pagerank值写入文件即可。

block stripe pagerank

代码实现

导入数据集

在读取data.txt的数据时，我们获取到其中的节点关系以及节点总数目。并且我们根据这些建立link_matrix矩阵。其每一个元素是一个列表，里面存储着对应节点的索引、出度还有到达的所有节点。其中到达的所有节点组成一个列表。然后我们对link_matrix依据对r的分组进行对应分组，然后将我们的分组写入对应的磁盘进行保存。

构建link_matirx:

```
1 def load_data():
2     # Load the data from the file
3     global max_node_index
4     global Num
5     global nodes_set
6     link_matrix = [[i,0,[]] for i in range(max_node_index+1)] #link matrix
7     src degree dest[]
8     max_node_index=0
9     with open(data_path) as f:
10         lines = f.readlines()
11         for line in lines:
12             split_line = line.split()
13             fr, to = int(split_line[0]), int(split_line[1])
14             link_matrix[fr][1]+=1 #degree
15             link_matrix[fr][2].append(to) #dest
16             max_node_index = max(max_node_index,fr,to)
17         Num=max_node_index
18         #把link_matrix中出度为0的节点删除
19         link_matrix=[link_matrix[i] for i in range(max_node_index+1) if
20 link_matrix[i][1]>0]
21
22     ...
23
24     #保存到文件
25     for i in range(group_num):
26         with open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX,'wb')
27 as f:
28         for line in link_matrix_groups[i]:
29             pickle.dump(line,f)
```

对link_matrix进行分组:

```

1  ...
2  link_matrix_groups=[[i,0,[]] for i in range(max_node_index+1)] for j in
   range(group_num)]
3      #分块
4      for line in link_matrix:
5          for dest in line[2]:
6              group_id=get_group_id(dest)
7              link_matrix_groups[group_id][line[0]][1]=line[1] #注意这里的degree是
   一样的
8              link_matrix_groups[group_id][line[0]][2].append(dest)
9      #把每个group中的出度为0的节点删除
10     for i in range(group_num):
11         link_matrix_groups[i]=[link_matrix_groups[i][j] for j in
   range(max_node_index+1) if link_matrix_groups[i][j][1]>0]
12  ...

```

分组写入磁盘：

```

1      #保存到文件
2      for i in range(group_num):
3          with open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX,'wb') as
   f:
4              for line in link_matrix_groups[i]:
5                  pickle.dump(line,f)

```

分块r_new计算

我们对r_new进行分块更新计算，每次我们依据r_new当前的块号来寻找到存储M的文件，从里面逐行读取，依据读取的信息再读磁盘寻找对应的src节点的Pagerank值。

我们会用一个列表来对每块的r_new的和值进行存储，这样我们对所有块的r_new进行初步更新后便获得了s的值。然后便可以分块进行归一化操作。

具体的代码如下：

```

1  def compute_rnew(flag):
2      sum_r_group=np.zeros(group_num)
3      for i in range(group_num):
4          #initialize r_new_stripe
5          r_new_stripe =np.zeros(get_group_size())
6          with open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX,'rb')
   as f1:
7              while True:
8                  try:
9                      # 逐行读取
10                     row_data = pickle.load(f1)
11                     # print(row_data)
12                     src, degree, dest = row_data

```



```

13         #读出r_old[src]
14         r_index=get_group_id(src)
15         if flag==0:
16             with
open(R_VECTOR_PATH_PREFIX+str(r_index)+R_VECTOR_PATH_SUFFIX,'rb') as f_r:
17                 r_tmp_stripe=pk1.load(f_r)
18         else:
19             with
open(R_NEW_VECTOR_PATH_PREFIX+str(r_index)+R_NEW_VECTOR_PATH_SUFFIX,'rb') as
f_r:
20                 r_tmp_stripe=pk1.load(f_r)
21                 for k in range(len(dest)):
22                     dest_index=dest[k]%get_group_size()
23                     r_new_stripe[dest_index] += belta *
r_tmp_stripe[src%get_group_size()] / degree
24             except EOFError:
25                 break # 如果到达文件末尾, 跳出循环
26             sum_r_group[i]=np.sum(r_new_stripe)
27             #保存r_new至磁盘上
28             if flag==0:
29                 with
open(R_NEW_VECTOR_PATH_PREFIX+str(i)+R_NEW_VECTOR_PATH_SUFFIX,'wb') as fr:
30                     pk1.dump(r_new_stripe,fr)
31             else:
32                 with open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX,'wb') as
fr:
33                     pk1.dump(r_new_stripe,fr)
34             return np.sum(sum_r_group)

```

分块r_new进行归一化

当我们获取到s值后, 然后我们对r_new进行分块归一化操作, 并在里面计算error, 来为后面的收敛条件判断做准备。

具体代码如下:

```

1 def normalize_r_new(flag,sum):
2     error=0.0
3     for i in range(group_num):
4         if flag==0:
5             with
open(R_NEW_VECTOR_PATH_PREFIX+str(i)+R_NEW_VECTOR_PATH_SUFFIX,'rb') as f:
6                 r_new_stripe=pk1.load(f)
7                 with open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX,'rb') as
f1:
8                     r_old_stripe=pk1.load(f1)
9             else:
10                 with open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX,'rb') as
f:
11                     r_new_stripe=pk1.load(f)

```

```

12         with
13         open(R_NEW_VECTOR_PATH_PREFIX+str(i)+R_NEW_VECTOR_PATH_SUFFIX, 'rb') as f1:
14             r_old_stripe=pk1.load(f1)
15             r_new_stripe+=np.ones(get_group_size()*(1-sum)/Num
16             if i==0:
17                 r_new_stripe[0] = 0.0
18             if i==group_num-1:
19                 r_new_stripe[get_last_group_size():] = [0]*(get_group_size()-
20                 get_last_group_size())
21                 error+=np.sum(np.abs(r_new_stripe-r_old_stripe))
22                 if flag==0:
23                     with
24                     open(R_NEW_VECTOR_PATH_PREFIX+str(i)+R_NEW_VECTOR_PATH_SUFFIX, 'wb') as f2:
25                         pk1.dump(r_new_stripe, f2)
26                     else:
27                         with open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX, 'wb') as
28                         f2:
29                             pk1.dump(r_new_stripe, f2)
30         return error

```

收敛条件判断

收敛条件的判断与basic Pagerank算法一致，不再赘述。

最终结果写入文件

最终结果写入文件与**basic Pagerank**基本一致，只不过此时是分块进行。将一个块更新之后，我们从里面挑选出排名前100的节点的索引和Pagerank值，将其加入到result列表中。待所有块都更新完之后，以及最后达到收敛情况后，我们在对result列表进行排序，从中间挑选前100的写入磁盘即可。

```

1  def print_result():
2      #降序排列
3      results=[]
4      #先从磁盘读取r_stripe
5      for i in range(group_num):
6
7          r_stripe=pk1.load(open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX, 'rb'))
8          #保存top100至results
9          index = np.argsort(-r_stripe)
10         for j in range(OUTPUT_NUM):
11             results.append((index[j]+i*get_group_size(),r_stripe[index[j]]))
12         #top100排序
13         results.sort(key=lambda x:x[1],reverse=True)
14         #output the result
15         with open(RESULT_PATH, "w") as f:
16             for i in range(OUTPUT_NUM):
17                 f.write(f"{results[i][0]} {results[i][1]}\n")

```

block stripe pagerank加速算法

代码实现

导入数据集

加速算法中导入数据集与**block stripe pagerank**算法中不一样的便是，此时我们的M是可以分块读取的，而不是按行读取。所以我们把M分块写入时我们是直接将一整块的内容导入磁盘，而不是一行行写入磁盘。

block stripe pagerank算法如下：

```
1  #保存到文件
2      for i in range(group_num):
3          with open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX, 'wb') as
f:
4              for line in link_matrix_groups[i]:
5                  pickle.dump(line, f)
```

block stripe pagerank加速算法如下：

```
1  #保存到文件
2      for i in range(group_num):
3          with open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX, 'wb') as
f:
4              pickle.dump(link_matrix_groups[i], f)
```

分块r_new计算

依据我们上述的思想，我们加速算法计算r_new如下：

```
1  def compute_rnew(flag):
2      sum_r_group=np.zeros(group_num)
3      for i in range(group_num):
4          #initialize r_new_stripe
5          r_new_stripe =np.zeros(get_group_size())*1.0
6          f1 = open(LINK_MATRIX_PATH_PREFIX+str(i)+LINK_MATRIX_PATH_SUFFIX, 'rb')
7          # 获取矩阵
8          matrix_stripe = pickle.load(f1)
9          # r_old_stripe
10         for j in range(group_num):
11             # src列表获取
12             src_index_in_matrix = [idx for idx, elem in
enumerate(matrix_stripe) if get_group_size()*j<= elem[0] < get_group_size()*
(j+1)]
13             if flag==0:
```

```

14         with
15         open(R_VECTOR_PATH_PREFIX+str(j)+R_VECTOR_PATH_SUFFIX, 'rb') as f_r:
16             r_tmp_stripe=pk1.load(f_r)
17         else:
18             with
19             open(R_NEW_VECTOR_PATH_PREFIX+str(j)+R_NEW_VECTOR_PATH_SUFFIX, 'rb') as f_r:
20                 r_tmp_stripe=pk1.load(f_r)
21             for src_index in src_index_in_matrix:
22                 index = matrix_stripe[src_index][0]%get_group_size()
23                 degree = matrix_stripe[src_index][1]
24                 des_list = matrix_stripe[src_index][2]
25                 for k in range(len(des_list)):
26                     dest_index=des_list[k]%get_group_size()
27                     r_new_stripe[dest_index] += belta * r_tmp_stripe[index] /
28                 degree
29             # sum计算
30             sum_r_group[i]=np.sum(r_new_stripe)
31             #保存r_new至磁盘上
32             if flag==0:
33                 with
34                 open(R_NEW_VECTOR_PATH_PREFIX+str(i)+R_NEW_VECTOR_PATH_SUFFIX, 'wb') as fr:
35                     pk1.dump(r_new_stripe, fr)
36             else:
37                 with open(R_VECTOR_PATH_PREFIX+str(i)+R_VECTOR_PATH_SUFFIX, 'wb') as
38                 fr:
39                     pk1.dump(r_new_stripe, fr)
40             return np.sum(sum_r_group)

```

如上，可以看到我们避免了r的pagerank块的重复读取，每当我们读取一个r的块，就会解决掉每一块M中与其相关的r_new值的更新。这样大大提高了计算速度和效率。

结果分析

basic pagerank

我们首先运行我们编写的**basic pagerank**算法，得到如下结果：

The No.65 node is 6005, and the pagerank is 0.0006961435
The No.66 node is 6190, and the pagerank is 0.0006958652
The No.67 node is 5655, and the pagerank is 0.0006951709
The No.68 node is 251, and the pagerank is 0.0006941162
The No.69 node is 3951, and the pagerank is 0.0006932836
The No.70 node is 8018, and the pagerank is 0.0006927310
The No.71 node is 233, and the pagerank is 0.0006920899
The No.72 node is 2589, and the pagerank is 0.0006915669
The No.73 node is 5996, and the pagerank is 0.0006914655
The No.74 node is 482, and the pagerank is 0.0006911669
The No.75 node is 972, and the pagerank is 0.0006904394
The No.76 node is 7499, and the pagerank is 0.0006866889
The No.77 node is 7442, and the pagerank is 0.0006863674
The No.78 node is 1173, and the pagerank is 0.0006859061
The No.79 node is 2369, and the pagerank is 0.0006853189
The No.80 node is 6315, and the pagerank is 0.0006836441
The No.81 node is 5129, and the pagerank is 0.0006834443
The No.82 node is 7784, and the pagerank is 0.0006832341
The No.83 node is 5998, and the pagerank is 0.0006830155
The No.84 node is 4692, and the pagerank is 0.0006828100
The No.85 node is 4255, and the pagerank is 0.0006827737
The No.86 node is 6692, and the pagerank is 0.0006823346
The No.87 node is 4832, and the pagerank is 0.0006819588
The No.88 node is 5275, and the pagerank is 0.0006802015
The No.89 node is 5376, and the pagerank is 0.0006796005
The No.90 node is 2232, and the pagerank is 0.0006776274
The No.91 node is 6928, and the pagerank is 0.0006768902
The No.92 node is 260, and the pagerank is 0.0006756163
The No.93 node is 1677, and the pagerank is 0.0006755145
The No.94 node is 6847, and the pagerank is 0.0006739259
The No.95 node is 6883, and the pagerank is 0.0006734445
The No.96 node is 7702, and the pagerank is 0.0006734059
The No.97 node is 1798, and the pagerank is 0.0006725799
The No.98 node is 4681, and the pagerank is 0.0006715352
The No.99 node is 2664, and the pagerank is 0.0006707611

对应的结果文件中如下：

basic_block > ≡ result.txt


```
1 2730 0.0008718594611375559
2 7102 0.0008545340959002473
3 1010 0.0008496161574824608
4 368 0.0008359030326148368
5 1907 0.0008305946654101579
6 7453 0.0008206465013603185
7 4583 0.0008178829088515284
8 7420 0.0008103357990263446
9 1847 0.0008099989394341323
10 5369 0.000805999534222782
11 3164 0.0008050933159985215
12 7446 0.0008031631400231676
13 3947 0.00080222396913202
14 2794 0.0007923234206487083
15 3215 0.0007821727145393269
16 5346 0.000781201823351276
17 7223 0.0007773694036888383
18 630 0.0007743796228268197
19 4417 0.0007688212611599778
20 4955 0.0007607900999000982
21 3208 0.0007590296060578478
22 2902 0.000757542554344257
23 5671 0.0007558530160705183
24 5833 0.0007516621205210425
25 5553 0.0007479602932840966
26 8096 0.000747405004014917
27 3204 0.0007457712158569753
28 758 0.0007449424434369266
29 6301 0.0007445833195291855
```

block stripe Pagerank

我们运行我们编写的block stripe Pagerank算法，然后得到如下结果：

```
error: 1.8734915628548297e-05
error: 1.4960354844896772e-05
error: 1.1946262343640895e-05
error: 9.539425064792824e-06
error: 7.61749808809344e-06
error: 6.08278556967121e-06
error: 4.857274639930458e-06
error: 3.878669839349094e-06
error: 3.0972264980897414e-06
error: 2.4732221037921737e-06
error: 1.974937118294536e-06
error: 1.5770426017498683e-06
error: 1.2593126864317696e-06
error: 1.005596450065954e-06
error: 8.02996929396081e-07
error: 6.412155379591808e-07
error: 5.120285658656877e-07
error: 4.088691505481571e-07
error: 3.264934677626747e-07
error: 2.60714178380457e-07
error: 2.081875733406306e-07
error: 1.6624360794055353e-07
error: 1.32750177130686e-07
error: 1.060047345489027e-07
error: 8.464774915930434e-08
error: 6.759359866038994e-08
error: 5.3975381702605556e-08
error: 4.31008538278723e-08
error: 3.441723922970475e-08
error: 2.748312911352301e-08
error: 2.1946048336781907e-08
error: 1.7524534401668632e-08
error: 1.3993831590748261e-08
error: 1.1174466347405813e-08
error: 8.923124377215594e-09
```

对应的结果文件如下：

block_stripe > result >  result.txt

```
1 2730 0.0008718594472507379
2 7102 0.0008545340821741086
3 1010 0.0008496161438322781
4 368 0.0008359030192162636
5 1907 0.000830594652057859
6 7453 0.0008206464885174505
7 4583 0.0008178828958575527
8 7420 0.0008103357860256772
9 1847 0.0008099989267240644
10 5369 0.0008059995215413375
11 3164 0.0008050933031909999
12 7446 0.0008031631271411682
13 3947 0.0008022239564882007
14 2794 0.0007923234080587147
15 3215 0.0007821727022003083
16 5346 0.0007812018110478533
17 7223 0.0007773693914095681
18 630 0.0007743796106612779
19 4417 0.0007688212491268002
20 4955 0.00076079008792649
21 3208 0.000759029593991941
22 2902 0.0007575425425289358
23 5671 0.0007558530042747153
24 5833 0.000751662108883537
25 5553 0.0007479602815379619
26 8096 0.0007474049923439749
27 3204 0.0007457712041696697
28 758 0.0007449424318682645
29 6301 0.000744583307897068
```

block stripe Pagerank加速算法

我们运行我们编写的block stripe Pagerank加速算法，得到的结果如下：


```
error: 2.346181405785621e-05
error: 1.8734915628548297e-05
error: 1.4960354844896772e-05
error: 1.1946262343640895e-05
error: 9.539425064792824e-06
error: 7.61749808809344e-06
error: 6.08278556967121e-06
error: 4.857274639930458e-06
error: 3.878669839349094e-06
error: 3.0972264980897414e-06
error: 2.4732221037921737e-06
error: 1.974937118294536e-06
error: 1.5770426017498683e-06
error: 1.2593126864317696e-06
error: 1.005596450065954e-06
error: 8.02996929396081e-07
error: 6.412155379591808e-07
error: 5.120285658656877e-07
error: 4.088691505481571e-07
error: 3.264934677626747e-07
error: 2.60714178380457e-07
error: 2.081875733406306e-07
error: 1.6624360794055353e-07
error: 1.32750177130686e-07
error: 1.060047345489027e-07
error: 8.464774915930434e-08
error: 6.759359866038994e-08
error: 5.3975381702605556e-08
error: 4.31008538278723e-08
error: 3.441723922970475e-08
error: 2.748312911352301e-08
error: 2.1946048336781907e-08
error: 1.7524534401668632e-08
error: 1.3993831590748261e-08
error: 1.1174466347405813e-08
error: 8.923124377215594e-09
```

对应的结果文件如下：

```
block_stripe > result > result.txt
1 2730 0.0008718594472507379
2 7102 0.0008545340821741086
3 1010 0.0008496161438322781
4 368 0.0008359030192162636
5 1907 0.000830594652057859
6 7453 0.0008206464885174505
7 4583 0.0008178828958575527
8 7420 0.0008103357860256772
9 1847 0.0008099989267240644
10 5369 0.0008059995215413375
11 3164 0.0008050933031909999
12 7446 0.0008031631271411682
13 3947 0.0008022239564882007
14 2794 0.0007923234080587147
15 3215 0.0007821727022003083
16 5346 0.0007812018110478533
17 7223 0.0007773693914095681
18 630 0.0007743796106612779
19 4417 0.0007688212491268002
20 4955 0.00076079008792649
21 3208 0.000759029593991941
22 2902 0.0007575425425289358
23 5671 0.0007558530042747153
24 5833 0.000751662108883537
25 5553 0.0007479602815379619
26 8096 0.0007474049923439749
27 3204 0.0007457712041696697
28 758 0.0007449424318682645
29 6301 0.000744583307897068
```

总结

经过测试，我们的basic pagerank算法得出结果的速度最快，block stripe Pagerank加速算法次之，block stripe pagerank算法最慢。同时我们注意到三种算法对最后节点的排名是相同的，但是score值具有一些精度上的小差异。

改进方向

当内存不足的情况下，我们会使用分块的算法来用时间换空间。但是不合理的分块会导致时间的大量浪费，同时某些块的重复读取会导致大量的磁盘I/O，使得性能大大降低。因此未来的方向便是分块时如何制定合理的分块策略，如何减少块的重复读取和时间浪费。