



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

大数据计算及应用期末报告

---

## PageRank 算法实现及优化

---

姜奕兵 1710218

皮春莹 1711436

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 4 月 14 日

## 摘要

基于 Python 实现了 PageRank 基本算法,对特殊情况的 dead ends 和 spider trap 进行处理,并对基本算法进行了稀疏矩阵优化实现了分块计算,给出提供数据集的 PageRank 结果(包括排名前 100 的 NodeID 及其 PageRank 分数)。

关键字: PageRank; Python

## 目录

一、 问题描述	1
(一) 实验环境	1
(二) 数据集描述	1
二、 代码实现细节	1
(一) 数据准备	1
(二) 分块 PageRank 实现	2
1. 稀疏矩阵优化	3
2. dead ends 处理	3
3. spider trap 处理	3
4. 分块计算	4
(三) 完整代码	4
三、 实验结果	7
(一) 优化性能对比	9

## 一、 问题描述

PageRank 算法实现及分块矩阵优化, 并根据给定 Wikidata.txt 数据集进行测试验证, 并输出 PageRank 值排名前 100 的 NodeID 及其 PageRank scores。

### (一) 实验环境

- 系统环境: Windows10 家庭中文版 (64 位), MacOS 10.15
- 编译环境: Python 3.7.4
- IDE: Jupyter, PyCharm

### (二) 数据集描述

数据集 Wikidata.txt 以 txt 格式存储了互相连接的节点, 每一行以制表符隔表示从 FromNodeID 到 ToNodeID 的连接。共有 103689 行数据, NodeID 取值范围为 [0,8927]。

数据集调用位置, 与源文件同一根目录即可。

## 二、 代码实现细节

### (一) 数据准备

首先将数据集按 3000 个为一组数据分块, 分别保存不同文件中。之后获取全部 NodeID 序列及其中 ID 最大节点以建立节点之间的映射关系。建立映射关系是为了节约内存空间, 因为有的 NodeID 实际上是不存在的。之后对 PageRank 向量进行初始化, 每个元素均初始化为  $\frac{1}{len(node)}$ 。再计算各个节点出度用以过滤处理 dead ends。

#### 数据准备

```
1 # 数据分块保存到 output, 返回分块文件数
2 def divide_data(self, block_size)->int:
3     inputfile = open(self.input_path, 'r')
4     count = 0
5     file_index = 0
6     dest_file = None
7     for line in inputfile:
8         if count % block_size == 0:
9             if dest_file:
10                 dest_file.close()
11                 dest_file = open(self.block_base_path + \
12                                 str(file_index) + '.txt', 'w')
13                 file_index += 1
14                 dest_file.write(line)
15                 count += 1
16     return file_index
17 # 获取数据中 Node、最大 NodeID
18 def get_Info(self, num):
19     node = []
20     max_node = 0
```

```

21     for i in range(num):
22         filename = self.block_base_path + str(i) + '.txt'
23         f = open(filename, 'r')
24         for line in f:
25             x, y = line.split()
26             max_node = max(max_node, max(int(x), int(y)))
27             node.append(int(x))
28             node.append(int(y))
29         f.close()
30     node = list(set(node)) # 去重
31     return node, max_node
32 # 节点映射及r向量初值
33 def preprocess(self, node, max_node):
34     nodemap = [0 for i in range(max_node + 1)]
35     for i in range(len(node)):
36         nodemap[node[i]] = i
37     # 初始化page_rank向量
38     r = [1.0 / len(node) for i in range(len(node))]
39     return nodemap, r
40 # 各节点出度
41 def get_out_degree(self, num, node, nodemap):
42     out = [0 for i in range(len(node))]
43     for i in range(num):
44         block = self.block_base_path + str(i) + '.txt'
45         f = open(block, 'r')
46         for line in f:
47             index = int(line.split()[0])
48             index = nodemap[index]
49             out[index] += 1
50     return out

```

## (二) 分块 PageRank 实现

根据 Basic 的 PageRank 算法, 在内存足够的情况下, 只要通过矩阵乘法在达到阈值之前的反复迭代, 本质为马尔科夫链模型, 就可以得到最终结果。但仍有一些特殊情况需要我们区别对待——dead ends 及 spide trap 修正。

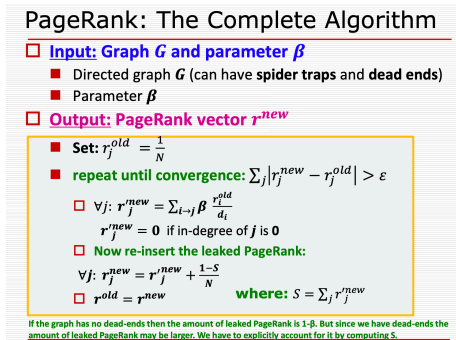


图 1: PageRank Algorithm

## 1. 稀疏矩阵优化

首先要对稀疏矩阵进行优化, 按照分块数据读取后, 只存储那些非零的链接。

稀疏矩阵优化

```

1  # 以稀疏形式保存, 只存储非零链接
2  def get_edges(self, data, nodemap):
3      data = np.array(data)
4      edges = np.zeros(data.shape)
5      for i in range(len(data)):
6          edges[i] = [nodemap[data[i][0]], nodemap[data[i][1]]]
7      return edges

```

## 2. dead ends 处理

如图 m 就是一个 dead ends, 即那些出度为 0 不指向任何一个节点的节点。对于 dead ends 的处理是借助 random teleports 解决的, 即令 dead ends 指向任意一个节点。

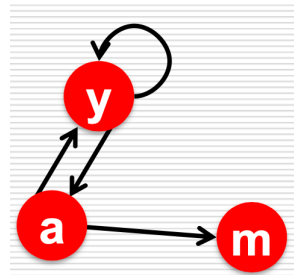


图 2: dead ends 示意图

dead ends

```

1  r_new[int(edge[1])] += r_old[int(edge[0])] * beta / out[int(edge[0])]

```

## 3. spider trap 处理

如图 y 就是一个 spider trap, 即那些自己指向自己的节点或多个节点。对于 dead end 进行 random teleports, 即解决了 spider trap 问题

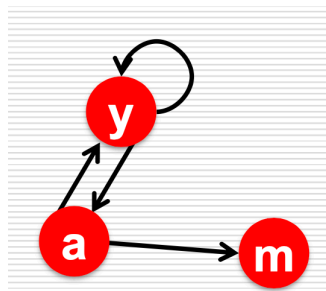


图 3: dead ends 示意图

#### 4. 分块计算

由于在矩阵计算过程中,有一部分计算是重复的,且为了节省内存。利用分块矩阵去修改它的值,会达到减少计算量加快计算效率的效果。

##### 分块计算

```

1      # 分块读取数据构成分块矩阵
2      for i in range(num):
3          block = self.block_base_path + str(i) + '.txt'
4          data = self.load_data(block)
5          edges = self.get_edges(data, nodemap)
6          for edge in edges:
7              r_new[int(edge[1])] += r_old[int(edge[0])] * beta / out[int(edge[0])]
8      r_sum = sum(r_new) # 对应相加得到一轮迭代的r向量
9      r_sub = np.ones(len(node)) * (1 - r_sum) / len(node) # rnew=rnew'+(1-S)/N
10     r_cur = r_new + r_sub

```

### (三) 完整代码

##### 完整代码

```

1      import numpy as np
2      import time
3      import psutil
4      import os
5
6      showMem = True
7
8      class page_rank():
9          def __init__(self):
10             self.input_path = 'WikiData.txt'
11             os.mkdir("./output/")
12             self.block_base_path = 'output/'
13             # 分块数据读取
14             def load_data(self, path):
15                 f = open(path, 'r')
16                 data = []
17                 for line in f:
18                     x=int(line.split()[0])
19                     y=int(line.split()[1])
20                     data.append([x, y])
21                 f.close()
22                 return data
23
24             # 以稀疏形式保存,只存储非零链接
25             def get_edges(self, data, nodemap):
26                 data = np.array(data)
27                 edges = np.zeros(data.shape)

```

```

28     for i in range(len(data)):
29         edges[i] = [nodemap[data[i][0]], nodemap[data[i][1]]]
30     return edges
31
32 # 数据分块保存到 output，返回分块文件数
33 def divide_data(self, block_size)->int:
34     inputfile = open(self.input_path, 'r')
35     count = 0
36     file_index = 0
37     dest_file = None
38     for line in inputfile:
39         if count % block_size == 0:
40             if dest_file:
41                 dest_file.close()
42                 dest_file = open(self.block_base_path + str(file_index) + \
43                                 '.txt', 'w')
44                 file_index += 1
45                 dest_file.write(line)
46                 count += 1
47     return file_index
48
49 # 获取数据中 Node、最大 NodeID
50 def get_Info(self, num):
51     node = []
52     max_node = 0
53     for i in range(num):
54         filename = self.block_base_path + str(i) + '.txt'
55         f = open(filename, 'r')
56         for line in f:
57             x, y = line.split()
58             max_node = max(max_node, max(int(x), int(y)))
59             node.append(int(x))
60             node.append(int(y))
61         f.close()
62     # 去重
63     node = list(set(node))
64     return node, max_node
65
66 # 节点映射及 r 向量初值
67 def preprocess(self, node, max_node):
68     nodemap = [0 for i in range(max_node + 1)]
69     for i in range(len(node)):
70         nodemap[node[i]] = i
71     # 初始化 page_rank 向量
72     r = [1.0 / len(node) for i in range(len(node))]
73     return nodemap, r
74
75 # 各节点出度

```

```

76 def get_out_degree(self, num, node, nodemap):
77     out = [0 for i in range(len(node))]
78     for i in range(num):
79         block = self.block_base_path + str(i) + '.txt'
80         f = open(block, 'r')
81         for line in f:
82             index = int(line.split()[0])
83             index = nodemap[index]
84             out[index] += 1
85     return out
86
87 # pagerank
88 def block_stripe_page_rank(self, num, node, nodemap, r, out, beta, threshold):
89     r_old = r
90     while True:
91         r_new = [0 for i in range(len(node))]
92         # 分块读取数据构成分块矩阵
93         for i in range(num):
94             block = self.block_base_path + str(i) + '.txt'
95             data = self.load_data(block)
96             edges = self.get_edges(data, nodemap)
97             for edge in edges:
98                 r_new[int(edge[1])] += \
99                     r_old[int(edge[0])] * beta / out[int(edge[0])]
100         if showMem:
101             process = psutil.Process(os.getpid())
102             print('Used Memory:', process.memory_info().rss / 1024 / 1024, 'MB')
103             showMem = False
104         r_sum = sum(r_new) # 对应相加得到一轮迭代的r向量
105         # rnew=rnew'+(1-S)/N
106         r_sub = np.ones(len(node)) * (1 - r_sum) / len(node)
107         r_cur = r_new + r_sub
108         s = np.sqrt(sum((r_cur - r_old) ** 2)) # 验证是否到收敛
109         # 单次迭代改变值小于threshold时,判断收敛,结束循环
110         if s <= threshold:
111             r_old = r_cur
112             break
113         else:
114             r_old = r_cur
115     return r_old
116
117 # 获取top100ID及其pagerank值
118 def get_top(self, num, nodemap, r):
119     r_index = r.argsort()[::-1][:100]
120     r.sort()
121     top_r = r[::-1][:100]
122     top_index = np.zeros(100)
123     for i in range(100):

```



```

124         top_index[i] = nodemap.index(r_index[i])
125     top_index = [int(i) for i in top_index]
126     for i in range(100):
127         print(top_index[i], top_r[i])
128     return top_index, top_r
129
130     # 写结果
131     def write_result(self, top_index, top_r):
132         f = open('result.txt', 'w')
133         for i in range(len(top_index)):
134             f.write(str(top_index[i]) + ' ' + str(top_r[i]) + '\n')
135         f.close()
136         return
137
138     # 运行
139     def do_page_rank(self):
140         block_num = self.divide_data(3000) # 切分数据
141         node, max_node = self.get_Info(block_num) # Node整理 MaxNodeID
142         nodemap, r = self.preprocess(node, max_node) # 稀疏优化
143         out = self.get_out_degree(block_num, node, nodemap) # 出度
144         # 执行PR
145         r_final = \
146         self.block_stripe_page_rank(block_num, node, nodemap, r, out, 0.85, 1e-8)
147         top_index, top_r = self.get_top(100, nodemap, r_final) # 输出结果
148         self.write_result(top_index, top_r) # 写文件
149
150
151 if __name__ == '__main__':
152     pr = page_rank()
153     start = time.clock()
154     pr.do_page_rank()
155     end = time.clock()
156     print('time cost: ', str(end - start), 's')

```

### 三、 实验结果

$\beta$  设置为 0.85, threshold 设置为  $10^{-8}$  的实验结果如下:

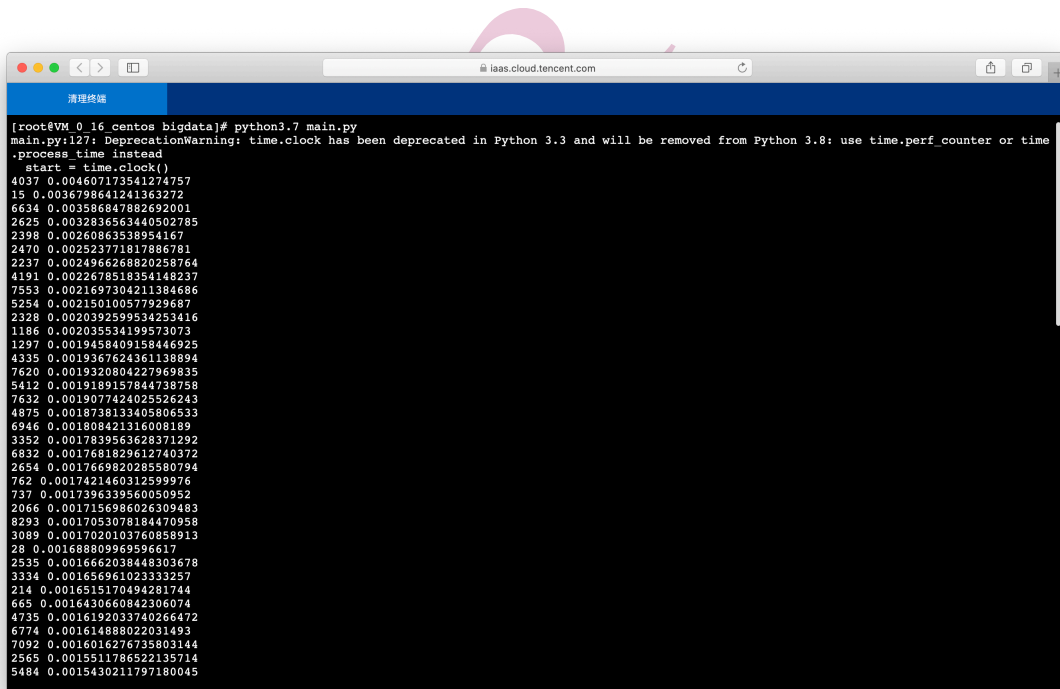
```

4037 0.004607173541274757
15 0.0036798641241363272
6634 0.003586847882692001
2625 0.0032836563440502785
2398 0.00260863538954167
2470 0.002523771817886781
2237 0.0024966268820258764
4191 0.0022678518354148237
7553 0.0021697304211384686
5254 0.002150100577929687
2328 0.0020392599534253416
1186 0.002035534199573073
1297 0.0019458409158446925
4335 0.0019367624361138894
7620 0.0019320804227969835
5412 0.0019189157844738758
7632 0.0019077424025526243
4875 0.0018738133405806533
6946 0.001808421316008189
3352 0.0017839563628371292
6832 0.0017681829612740372
2654 0.0017669820285580794
762 0.0017421460312599976
737 0.0017396339560050952
2066 0.0017156986026309483
8293 0.0017053078184470958
3089 0.0017020103760858913
28 0.001688809969596617
2535 0.0016662038448303678
3334 0.001656961023333257
214 0.0016515170494281744
665 0.0016430660842306074
4735 0.0016192033740266472
6774 0.001614888022031493
7092 0.0016016276735803144
2565 0.0015511786522135714
5484 0.0015430211797180045
8042 0.001476800977222003
4310 0.0014619977004505108
5423 0.001417625426080946
1211 0.00141758059627074
3456 0.0014168671928819546
2657 0.001365191656323992
5484 0.0013639541968892941
5232 0.0013617836226362246
4712 0.0013418884950684629
271 0.001326259372052897
4828 0.0013006247891935228
5879 0.0012991644029386336
4261 0.0012857557179177502

```

图 4: 实验结果  $\beta = 0.85$ (前 50 个)

云主机运行截图如下:



```

[root@VM 0 16_centos bigdata]# python3.7 main.py
main.py:127: DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be removed from Python 3.8: use time.perf_counter or time
.process_time instead
start = time.clock()
4037 0.004607173541274757
15 0.0036798641241363272
6634 0.003586847882692001
2625 0.0032836563440502785
2398 0.00260863538954167
2470 0.002523771817886781
2237 0.0024966268820258764
4191 0.0022678518354148237
7553 0.0021697304211384686
5254 0.002150100577929687
2328 0.0020392599534253416
1186 0.002035534199573073
1297 0.0019458409158446925
4335 0.0019367624361138894
7620 0.0019320804227969835
5412 0.0019189157844738758
7632 0.0019077424025526243
4875 0.0018738133405806533
6946 0.001808421316008189
3352 0.0017839563628371292
6832 0.0017681829612740372
2654 0.0017669820285580794
762 0.0017421460312599976
737 0.0017396339560050952
2066 0.0017156986026309483
8293 0.0017053078184470958
3089 0.0017020103760858913
28 0.001688809969596617
2535 0.0016662038448303678
3334 0.001656961023333257
214 0.0016515170494281744
665 0.0016430660842306074
4735 0.0016192033740266472
6774 0.001614888022031493
7092 0.0016016276735803144
2565 0.0015511786522135714
5484 0.0015430211797180045

```

图 5: 运行

通常情况下  $\beta$  的取值在  $[0.8, 0.9]$  之间, 下为设置不同  $\beta$  值进行计算的结果:

根据上述结果可以发现, 对不同的  $\beta$  值而言, 所得到的排名和 score 会有所不同, 但是其对排名的差异影响是很小的。而针对 spider trap 来说, 一个较好的  $\beta$  取值可以得到相对更好的 PageRank 结果。

```

4037 0.004515392297676418
15 0.0035416576372945493
6634 0.0032585902780373826
2625 0.0031114487075271655
2470 0.002530758832270017
2237 0.0024746214043402342
2398 0.0024472120347928633
4191 0.00216672164717081
5254 0.0020651957967971683
7553 0.002050300817012299
1186 0.0020336951615048788
2328 0.0019518007921207547
7620 0.00184345526482494
1297 0.0018376531188851815
4335 0.0018149671599272915
4875 0.0017985107889680549
7632 0.0017755316584507194
5412 0.0017724080620326883
2654 0.0017301268803569926
3352 0.001696720205395581
8293 0.0016896920464079962
6832 0.0016598874556481094
28 0.001654868676014266
762 0.0016542923163464667
665 0.0016469061231143161
6946 0.0016330677778249466
737 0.0016299360881053984
214 0.0016232765882327389
6774 0.0016044508580373928
2535 0.0015946392944650107
3089 0.001593905783339388
2066 0.0015919655454269357
3334 0.00157607747739779
4735 0.0015306708142179013
7092 0.0015119463586868546
2565 0.001476814684273226
5484 0.0014694536897782555
4310 0.0013673563233443514
5423 0.0013450103684866568
1211 0.001343376346439493
3456 0.0013207090272579155
8042 0.0013091882971984726
2657 0.0013041924572556297
271 0.0012868380669174824
5404 0.0012776418070001069
4261 0.001272568081548523
5233 0.0012694289004047796
2285 0.001257783354208203
4712 0.001244339697405144
5210 0.0012307016158967445

```

图 6: 实验结果  $\beta = 0.8$ (前 50 个)

```

4037 0.004404424278732542
15 0.0033948182434175
6634 0.002961514390578057
2625 0.00293885333140555
2470 0.0025188020771486403
2237 0.002438942320790857
2398 0.0022892950177082574
4191 0.002064085202698976
1186 0.0020187644154176427
5254 0.001976538715770022
7553 0.0019282952068597958
2328 0.0018621740396892032
7620 0.001754705408579599
1297 0.001729155419299459
4875 0.0017214540896452568
4335 0.0016964759656779458
2654 0.0016866108761733802
8293 0.0016638843331711166
7632 0.0016470358055569313
665 0.001638931442346041
5412 0.001632584969660807
28 0.0016118067532753157
3352 0.0016085084539888086
214 0.0015872477957034686
6774 0.0015853991016257728
762 0.0015655697784966991
6832 0.0015531296749494556
737 0.001522452117408981
2535 0.001521431583622326
3334 0.0014945115344184275
3089 0.0014882298491659504
6946 0.0014750852919319106
2066 0.001472675476136278
4735 0.001442991401407684
7092 0.0014238228210438778
2565 0.0014022407193546487
5484 0.001395522486453016
4310 0.0012754450083582798
5423 0.001270913841804167
1211 0.0012674625131508547
2285 0.001264591733405399
4261 0.0012530366162079917
2657 0.0012425190227234619
271 0.0012418981010332648
3456 0.0012280195061775306
5404 0.0011933595219839341
1842 0.0011867063640727319
5233 0.0011800855233002634
5210 0.0011792345383541756

```

图 8: 实验结果  $\beta = 0.75$ (前 50 个)

```

4037 0.004680026036075425
6634 0.003952827063834458
15 0.003809417118144997
2625 0.0034556866447225765
2398 0.0027740130093072335
2237 0.0025049376299668565
2470 0.002497476032617863
4191 0.002367831459378756
7553 0.002286064271733019
5254 0.002231281010554805
2328 0.002124691989660887
5412 0.002072510268684018
4335 0.002062206999441542
1297 0.0020533730848795837
7632 0.002043845995943238
1186 0.002024133819636614
7620 0.0020180835292312304
6946 0.00200465454187407
4875 0.0019478034582037045
6832 0.0018781306603051686
3352 0.0018702485137558696
737 0.0018516498910390305
2066 0.0018439252431276653
762 0.001828993127434074
3089 0.001812674393862785
2654 0.0017972194441720252
3334 0.0017373528654671512
2535 0.001736213389974111
28 0.0017128349792512594
8293 0.0017105792446239659
4735 0.0017087119315775277
7092 0.0016922662799605844
214 0.0016716761279041714
8042 0.001667385078756542
665 0.001627055992240785
2565 0.0016255592248908362
6774 0.001616890961608558
5484 0.0016164646004205004
4310 0.0015596245799508256
3456 0.0015167630780138128
1211 0.0014896243453749792
5423 0.0014885932562137697
5233 0.0014583242843442484
5404 0.001454209658339244
4712 0.0014448704222693497
8163 0.0014354983272661498
2657 0.0014256954966389928
4828 0.0014070023581304026
5079 0.0013729849946466858

```

图 7: 实验结果  $\beta = 0.9$ (前 50 个)

```

4037 0.0035498836311265245
15 0.002530993599427366
2470 0.0021826747181576396
2625 0.0020615259301986883
2237 0.0020524758912999054
6634 0.0017919559780663446
1186 0.0017476861893303613
2398 0.0015395350445383477
4191 0.0015172247585692803
5254 0.0014751349288587901
665 0.0014286888142562098
8293 0.001386684344024933
2328 0.001375860452496896
2654 0.0013657635207482716
6774 0.001354903650293517
7553 0.001297834520576714
4875 0.0012975621997176432
214 0.0012955971572663658
28 0.0012793862769531028
7620 0.0012746419621173601
1297 0.0011890334316446453
2285 0.0011547161356464837
3352 0.001151487533788772
4335 0.001143335301082835
2535 0.0011269374080080823
762 0.0011111967006788535
3334 0.0010723359693113352
4261 0.0010582765673998005
7632 0.0010571236676612415
6832 0.0010410486570865268
5412 0.0010218012142220224
2565 0.0010194040357836603
737 0.0010159839657828068
5484 0.0010142653017867602
4735 0.0010127532428444372
3089 0.000992680565442002
7092 0.000986022489412765
1842 0.0009662598021276589
271 0.0009510803867681319
2066 0.0009394348748730147
2657 0.0009184275943358693
1855 0.0008994798335132324
2643 0.0008945538390944304
5210 0.0008919435784299135
5423 0.0008852753187678957
1211 0.0008746978970185792
6946 0.0008665675710963629
4310 0.0008509058682853592
299 0.0008313233814682901

```

图 9: 实验结果  $\beta = 0.5$ (前 50 个)

## (一) 优化性能对比

此外我们对于稀疏矩阵优化以及分块矩阵计算做了时间和内存上的对比, 得到以下结果: (基本算法实现文件为 basic.py, main.py 为最终优化算法)

<i>Time</i>		稀疏优化	
		是	否
分块优化	是	7.36s	(未)
	否	7.26s	385.56s

表 1: 时间对比

<i>Memory</i>		稀疏优化	
		是	否
分块优化	是	32.96MB	(未)
	否	50.11MB	552.19MB

表 2: 空间对比

由以上两种对比方式可以看出, 尽管对于这样较小的测试集,basic 算法也要经过  $8297 * 8297$  次迭代才可以完成, 而这么大的运算量是没必要的, 所以在经过稀疏化处理后, 时间得到了极大地改善(加速比 50 左右), 同时稀疏矩阵的处理也对空间进行了优化; 而分块处理则是用少量时间换空间, 所以空间上还会有进一步的改善。

## 参考文献

- [1] 课件链接分析\_第二部分