

《数据科学与工程算法》项目报告

报告题目： 基于局部敏感哈希的图节点相似性查询系统

姓 名： 温兆和

学 号： 10205501432

完成日期： 2023.04.28

摘要 [中文]:

局部敏感哈希是最为著名的寻找相似集合的算法之一，它的基本思想就是用一组哈希函数求出集合的哈希值，把哈希签名矩阵分块，并根据每一个分块中两个集合的哈希签名向量是否相等来判断这两个集合是否相似。在多探寻局部敏感哈希算法中，我们可以进一步放宽集合相似的标准，将哈希签名向量相似的集合判断为相似，让更多的集合称为目标集合的备选相似集合，从而提高算法的准确性。

Abstract [English]

Locality-Sensitive Hashing (LSH) is one of the most popular algorithms on similarity search. Its basic idea is to work out the hash values of the set on a group of hash functions, divide the hash value matrix into several parts and decide whether two sets are similar by comparing their hash value vectors in each part of the hash value matrix. In Multi-Probe Locality-Sensitive Hashing, the standard of the similarity between two sets is further loosen: two sets can be judged as 'similar' if their hash value vectors are similar with each other so that more sets can be selected to the candidate group of the goal set, thus enhancing the accuracy of the algorithm.

一、项目概述（阐明项目的科学价值与相关研究工作，描述项目主要内容）

1.1 项目的科学价值

在本次实验中，数据集是表示研究人员之间共同作者关系的无向图。我们需要构建一个 LSH 方案，争取在更短的时间内返回邻居集合与输入节点的邻居集合 Jaccard 相似度最高的十个节点（不包括输入节点自身）。这样做的意义在于，在海量高维数据的前提下牺牲一定的准确率，降低数据的维度，提升查询的速度。

1.2 项目使用的算法及其相关研究工作

在课堂上，我们已经学习了最传统的 LSH 算法，即用多个哈希函数求出集合的哈希值，再对哈希签名矩阵进行分组，如果在任何一个分组中某个集合的哈希签名向量与目标集合的哈希签名向量相等，我们就认为这个集合与目标集合相似。这里的哈希函数是这样的：先把集合表示为一个只有 0 和 1 的向量，如果集合中有全集中的某个元素，那么相应元素就是 1，否则就是 0。接着对集合向量的行索引进行重排列，并将第一个不为 0 的行索引值作为集合的哈希值。近年来，有不少学术研究对传统 LSH 算法作出了改进：有些改进是针对哈希函数的；也有些改进，如多探寻局部敏感哈希¹，则是通过放大相似集合的范围来提升查询的准确率。

1.3 项目主要内容

在本次实验中，我们首先对数据集进行预处理，随机生成尽可能多的哈希函数并求出每个集合的哈希值；然后，实现传统 LSH 算法并测试其运行时间和准确度；最后，实现多探寻 LSH 并通过比较探究它的性能（时间、空间、准确度）是否比传统 LSH 算法有所改进。

二、问题定义（提供问题定义的语言描述与数学形式）

2.1 语言描述：

最原始的数据集是这样的：每一行有两个数字，每个数字代表图中的一个节点，每一行代表两个节点间有一条边。对于一个节点来说，所有跟它有边的节点构成了它的邻居集合。对于我们要查询的节点，算法应该返回所有节点中邻居集合与查询的节点的 Jaccard 相似度最高的十个节点（不包括查询节点自身）。

2.2 数学形式：

给定图

$$G = (V, E)$$

输入

$$node \in V$$

其中节点 i 的邻居集合与查询节点的邻居集合的 Jaccard 相似度定义为

$$N_i = \text{Jaccard}(\text{neighbour}(node), \text{neighbour}(i))$$

需要返回数列 $\{N_i\}$ 中最大的十个数字的索引。

三、方法（问题解决步骤和实现细节）

¹ Wei Dong. Zhe Wang. William Josephson. Moses Charikar. Kai Li. Modeling LSH for Performance Tuning

3.1 数据集预处理

该部分的所有工作均在 lab1_dataprocess.ipynb 文件中体现。

首先，逐行读取文件 ca-AstroPh.txt 中的数据，将其存入列表 data 中并将所有数据的类型从字符串转换为整型：

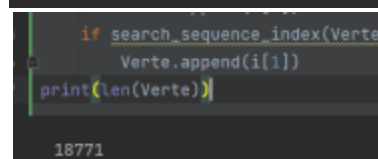
```
data = []
file = open('ca-AstroPh.txt', 'r')
file_data = file.readlines()
for i in range(1, len(file_data)):
    tmp_list = file_data[i].split(' ')
    tmp_list[-1] = tmp_list[-1].replace('\n', '')
    data.append(tmp_list)
for i in data:
    i[0] = int(i[0])
    i[1] = int(i[1])
```

利用 Python 中的 networkx 工具创建无向图 Charistic_Matrix，并将 data 列表中的边逐一添加到这个无向图中：

```
Characteristic_Matrix = nx.Graph()
for i in data:
    Characteristic_Matrix.add_edge(i[0]-1, i[1]-1)
```

通过读取无向图中的节点，我们可以发现这张图中有 18771 个节点：

```
Verte = []
for i in data:
    if search_sequence_index(Verte, i[0]) == False:
        Verte.append(i[0])
    if search_sequence_index(Verte, i[1]) == False:
        Verte.append(i[1])
print(len(Verte))
```



```
if search_sequence_index(Verte, i[0]) == False:
    Verte.append(i[0])
if search_sequence_index(Verte, i[1]) == False:
    Verte.append(i[1])
print(len(Verte))

18771
```

接着，我们需要构造尽可能多的哈希函数以对集合特征矩阵的行进行重排列。我们采用线性哈希函数，即

$$h(x) = (ax + b) \bmod 18771$$

其中一次项系数 a 必须与 18771 互素，常数项可以从 0-18770 之间随机生成。经过排查，我们发现 1-18770 之间共有 12512 个数字与 18771 互素，也就得到了 12512 个哈希函数：

```
def gcd(a, b):
    if (a==1&b==1):          # 两个正整数中，只有其中一个数值为 1，两个正整数为互质数
        return True
    while True:              # 求出两个正整数的最大公约数
        t = a%b
        if (t == 0):
            break
```

```

        else:
            a = b
            b = t
    if (b>1):
        return False# 如果最大公约数大于 1, 表示两个正整数不互质
    else:
        return True# 如果最大公约数等于 1, 表示两个正整数互质
Hash_Function = []
for i in range (len(Verte)):
    if (gcd(i+1,18771)):
        h_f = []
        normal_number = random.randint(0,18770)
        h_f.append(i+1)
        h_f.append(normal_number)
        Hash_Function.append(h_f)
print(len(Hash_Function))

```

```

7      h_f.append(normal_number)
8      Hash_Function.append(h_f)
9      print(len(Hash_Function))

```

12512

接着，我们就能计算每个集合在这 12512 个哈希函数下的哈希值了。计算的方法是，从刚刚定义的无向图中读取每一个节点的邻居集合，将这个集合和某个哈希函数传入 `find_msh()` 函数中后对集合中的每一个数字用传入的哈希函数作一次变换，最后返回变换后的所有数字中的最小值。

```

def find_msh(neighbours,h_f):
    hash_values = []
    for i in neighbours:
        hash_values.append((i*h_f[0]+h_f[1])%18771)
    hash_values.sort()
    return hash_values[0]
MSH = np.zeros((len(Verte),len(Hash_Function)))
for i in range (len(Verte)):
    for j in range (len(Hash_Function)):
        neighbours = list(nx.all_neighbors(Characteristic_Matrix, i))
        MSH[i][j]=find_msh(neighbours,Hash_Function[j])

```

最后，把数据预处理过程中生成的数据集、哈希函数和哈希签名矩阵存入 csv 文件，以便后续使用：

```
data_in_frame = DataFrame(data)
data_in_frame.to_csv('data.csv',index=0)
Hash_Function_in_frame = DataFrame(Hash_Function)
Hash_Function_in_frame.to_csv('Hash_Function.csv',index=0)
MSH_in_frame = DataFrame(MSH)
MSH_in_frame.to_csv('MSH.csv',index=0)
```

3.2 普通局部敏感哈希的实现

该部分的所有工作均在 lab1_main.ipynb 文件中体现。

首先，打开上一步生成的数据集、哈希签名矩阵并将数据集中的点添加到无向图 Characteristic_Graph 中：

```
data = pd.read_csv("data.csv")
MSH = pd.read_csv("MSH.csv")
Characteristic_Graph = nx.Graph()
for i in range(len(data)):
    Characteristic_Graph.add_edge(data['0'][i]-1,data['1'][i]-1)
Connection_Matrix=np.array(nx.adjacency_matrix(Characteristic_Graph).todense())
Verte = list(Characteristic_Graph.nodes)
```

从所有 18771 个节点中任选五个作为本次实验的测试点：

```
test_node = []
for i in range (5):
    test_node+= [random.randint(0,18770)]
```

下面介绍普通 LSH 的实现思路。在函数 LSH 中，输入查询节点 node、行条数 b 和每个行条的行数 r。首先，根据 b、r 对哈希签名矩阵进行分块，然后遍历分块后的哈希签名矩阵，在每个行条中比较查询集合的签名向量和其它集合的签名向量。只要有一个行条中某个集合的签名向量与查询节点邻居集合的签名向量相等，就将其加入一个备选集。遍历完成后，调用 similarest_ten() 函数，从备选集中找出与查询集合 Jaccard 相似度最高的十个集合并返回。

```
def LSH(node,b,r):# b 个行条，每行 r 个哈希函数
    global MSH
    global G
    global Verte
    MSH_tobe_Verteilung = np.array(MSH.iloc[:, 0:b*r])
    volunteer = []
    MSH_Verteilung = np.array_split(MSH_tobe_Verteilung, b, axis=1)
    print("MSH_Verteilung")
    print(MSH_Verteilung[0][node])
    for i in range (len(MSH_Verteilung)):
        for j in range (len(Verte)):
            if
(difference_less_than_one(MSH_Verteilung[i][j],MSH_Verteilung[i][node])
==0) & (j!=node) & (j not in volunteer):
                volunteer.append(j)
    print("volunteer")
```

```
print(volunteer)
return similarest_ten(node,volunteer)
```

在 `similarest_ten()` 函数中，输入查询集合与备选集合。逐一计算查询集合与每个备选集合的 Jaccard 相似度并排序，返回 Jaccard 相似度最高的十个集合：

```
def similarest_ten(node,Verte):
    global Characteristic_Graph
    max_index = []
    Real_Jaccard=pd.DataFrame(columns=['node','jaccard'],dtype=float)
    for i in Verte:
        df_1 =
pd.DataFrame([[i,jaccard(list(nx.all_neighbors(Characteristic_Graph,
node)),list(nx.all_neighbors(Characteristic_Graph,
i))))],columns=['node','jaccard'],dtype=float)
        Real_Jaccard = pd.concat([Real_Jaccard,df_1], ignore_index=True)
        Real_Jaccard=Real_Jaccard.sort_values(by="jaccard" ,
ascending=False)
        print("Jaccard")
        if len(Real_Jaccard)>0:
            print(Real_Jaccard)
        Real_Jaccard_list = np.array(Real_Jaccard).tolist()
        max_ten = 10
        if max_ten>len(Real_Jaccard_list):
            max_ten=len(Real_Jaccard_list)
        for i in range(max_ten):
            max_index.append(Real_Jaccard_list[i][0])
        if (len(max_index)<10):
            while (len(max_index)<10):
                max_index.append(-1)
    return max_index
```

Jaccard 相似度的计算方法是：两个集合交集的元素个数除以两个集合并集中的元素个数：

```
def jaccard(A,B):
    bigcap = len(list(set(A) & set(B)))
    bigcup = len(list(set(A) | set(B)))
    jaccard = bigcap/bigcup
    return jaccard
```

这里我们还定义了一个 `difference_less_than_one()` 函数。它在 `LSH()` 函数中被调用，是用来比较两个集合的哈希签名向量的。具体来说，它返回两个向量对应元素之差的绝对值的总和。

```
def difference_less_than_one(a,b):
    return np.abs(a-b).sum()
```

3.3 多探寻局部敏感哈希的实现

该部分的所有工作均在 `lab1_optimization.ipynb` 文件中体现。

多探寻局部敏感哈希的思路是这样的：除了根目标节点邻居集合签名向量完全相

等的集合，签名向量相近的集合也被认为是相似的。比如，某个集合的签名向量是[5,6,9]，那么签名向量为[6,7,9]、[5,8,9]或[5,7,8]的集合也被认为与该集合相似。所以说，多探寻局部敏感哈希的实现与普通局部敏感哈希相比只作了一点点改变：在 LSH 函数中多输入一个参数 **bound**，只要两个哈希签名向量对应元素之差的绝对值小于 **bound** 就可以认为它们对应的两个集合相似。

```
def LSH(node,b,r,bound):# b 个行条，每行 r 个哈希函数
    global MSH
    global G
    global Verte
    MSH_tobe_Verteilung = np.array(MSH.iloc[:, 0:b*r])
    volunteer = []
    MSH_Verteilung = np.array_split(MSH_tobe_Verteilung, b, axis=1)
    print("MSH_Verteilung")
    print(MSH_Verteilung[0][node])
    for i in range (len(MSH_Verteilung)):
        for j in range (len(Verte)):
            if
            (difference_less_than_one(MSH_Verteilung[i][j],MSH_Verteilung[i][node
            ])<=bound) & (j!=node) & (j not in volunteer):
                volunteer.append(j)
    print("volunteer")
    print(volunteer)
    return similarest_ten(node,volunteer)
```

另外，我们在多探寻局部敏感哈希的实现中沿用前面抽取到的五个测试点。它们在 3.2 的最后被存入 csv 文件。

```
selected_node = pd.read_csv("selected_node.csv")
test_node = []
for i in range (5):
    test_node+= [int(selected_node['node'][i])]
```

四、 实验结果（验证提出方法的有效性和高效性）

4.1 普通局部敏感哈希的运行结果

这里我们进行了三组测试，分别控制了三个不同的变量： $b*r$ 、 b 和 r 。我们通过 `conduct()` 函数，分别计算局部敏感哈希与直接搜索最相近的集合的运行时间并计算正确率。正确率的计算方法是这样的：局部敏感哈希返回的十个节点组成集合 **A**，直接搜索得到的十个节点（正确情况）组成集合 **B**，用 **A** 与 **B** 交集的元素个数除以 **B** 中元素的个数（十个）就是 LSH 的准确率。值得一提的是，**A** 中有效元素的个数很可能少于十个，因为能够在 LSH 算法中进入候选集的节点就很有可能少于十个。

```
def conduct(b,r):
    global test_node
    global Verte
    df =
```



```

pd.DataFrame(columns=['node', 'accuracy', 'time', 'blank_control'], dtype
=float)
    for i in range (5):
        print("-----")
        print("i")
        print(i)
        t_a=time.perf_counter()
        A = LSH(test_node[i],b,r)
        t_b=time.perf_counter()
        Verte_without_i = list(Characteristic_Graph.nodes)
        if test_node[i] in Verte_without_i:
            Verte_without_i.remove(test_node[i])
        t_c=time.perf_counter()
        B = similarest_ten(test_node[i],Verte_without_i)
        t_d=time.perf_counter()
        accuracy = len(list(set(A) & set(B)))/len(B)
        run_time = t_b-t_a
        compare_time = t_d-t_c
        df1 =
pd.DataFrame([[test_node[i],accuracy,run_time,compare_time]],columns=
['node', 'accuracy', 'time', 'blank_control'], dtype=float)
        df = pd.concat([df,df1], ignore_index=True)
        print("A")
        print(A)
        print("B")
        print(B)
    print(df)

```

4.1.1 控制 $b \cdot r = 12512$

这里我们进行了四次实验，具体情况如下：

```

26 1 conduct(6256,2)
    18769 18770.0 0.000000
    18770 rows x 2 columns
    A
    [9528.0, 7065.0, 9531.0, 9537.0, 7063.0, 2464.0, 9539.0, 9535.0, 9533.0, 9532.0]
    B
    [7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
      node  accuracy      time  blank_control
0   1617.0        0.9  1133.046244      8.515367
1   2606.0        1.0  1941.778672      8.618361
2   8131.0        1.0   718.681154      8.060001
3  13588.0        1.0   725.205274      8.173565
4   9536.0        1.0   736.992815      8.077427

```

```
In 22 1 conduct(3128,4)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[9528.0, 9531.0, 9537.0, 7065.0, 9532.0, 9533.0, 9535.0, 9539.0, 7062.0, 9538.0]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	1.0	441.337044	9.604685
1	2606.0	1.0	417.922232	10.096952
2	8131.0	1.0	395.398334	9.472690
3	13588.0	0.7	392.771007	9.503786
4	9536.0	0.8	399.346558	9.684781

```
In 24 1 conduct(1564,8)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	188.660291	9.565585
1	2606.0	0.0	174.105192	8.662297
2	8131.0	0.5	174.758080	8.269868
3	13588.0	0.3	174.992833	8.198900
4	9536.0	0.0	172.899221	8.289413

```
In 25 1 conduct(782,16)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.6	85.571515	8.208882
1	2606.0	0.0	86.795195	8.563412
2	8131.0	0.1	87.259601	8.065334
3	13588.0	0.0	86.316925	8.165431
4	9536.0	0.0	87.925898	8.193756

我们可以看到，随着行条数 **b** 的减少，运行时间也逐步减少，但无论在何种情况下，LSH 的运行时间都远大于直接搜索的时间，原因在于直接搜索只调用了 `similarest_ten()` 函数，其中最多只有一重 `for` 循环；而 LSH 在调用 `similarest_ten()` 函数之前还要进行分桶操作，那里需要进行一个迭代次数为 18771×12512 的双重循环，这耗费了大量的时间。但实际上，LSH 还是在大大缩小了备选集范围的前

提下达到了较高的正确率。以 `conduct(6256,2)` 的第一个测试节点为例，它直接把备选集的大小从原本的 18770 缩小到了 436。如果集合所含有的不同元素再多一点（或者说集合特征向量的维数更大），或许 LSH 的性能就能得到更好的体现。

```
26 1 | conduct(6256,2)
    1 1827.0 0.701707
    10 1612.0 0.705882
    27 1623.0 0.701754
    .. ...
    225 4837.0 0.008000
    354 1420.0 0.007812
    329 4464.0 0.007634
    368 2545.0 0.007519
    280 4743.0 0.006329

[436 rows x 2 columns]
Jaccard
node jaccard
```

还值得注意的是，查询的正确率随着每个行条的行数 r 的增加而减少，这是因为两个集合被哈希到同一桶的概率随着 r 值的增加而减少。甚至当 r 比较大的时候，所有的节点无一进入备选集，导致得到的正确率为 0。又如，在 `conduct(1564,8)` 的第四个测试节点中，备选集中只有三个节点入选。尽管入选的三个节点都是正确的，但由于数量不足本次实验要求的十个，得到的正确率还是 0.3。

```
24 1 | conduct(1564,8)
    7570 7577.0 0.120000
    ... ...
    6263 13653.0 0.000000
    6262 13652.0 0.000000
    6261 13293.0 0.000000
    6260 13292.0 0.000000
    18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[2846.0, 2849.0, 12408.0, -1, -1, -1, -1, -1, -1, -1]
B
[16558.0, 2846.0, 2849.0, 12408.0, 9359.0, 12397.0, 7303.0, 11967.0, 2842.0, 3281.0]
```

4.1.2 控制 $r=2$

现在来探索控制每个行条的行数不变，改变行条数会对运行时间和正确率有什么影响。

```
26 1 conduct(6256,2)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[9528.0, 7065.0, 9531.0, 9537.0, 7063.0, 2464.0, 9539.0, 9535.0, 9533.0, 9532.0]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	1133.046244	8.515367
1	2606.0	1.0	1941.778672	8.618361
2	8131.0	1.0	718.681154	8.060001
3	13588.0	1.0	725.205274	8.173565
4	9536.0	1.0	736.992815	8.077427

```
h 33 1 conduct(2346,2)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[9528.0, 7065.0, 9531.0, 9537.0, 7063.0, 2464.0, 9539.0, 9535.0, 9533.0, 9532.0]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	390.324085	8.330031
1	2606.0	1.0	523.051092	8.486917
2	8131.0	1.0	274.141552	8.141178
3	13588.0	1.0	270.830200	8.158149
4	9536.0	1.0	271.954587	8.398401

```
27 1 conduct(1564,2)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[9528.0, 7065.0, 9531.0, 9537.0, 7063.0, 2464.0, 9539.0, 9535.0, 9533.0, 9532.0]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	250.190089	8.406944
1	2606.0	1.0	299.238387	8.614090
2	8131.0	1.0	177.864454	8.141581
3	13588.0	1.0	178.776733	8.256538
4	9536.0	1.0	177.061415	8.048621

```
32 1 conduct(782,2)

18/69 18770.0 0.000000

[18770 rows x 2 columns]
A
[9537.0, 9528.0, 9531.0, 2464.0, 7063.0, 9532.0, 9533.0, 9535.0, 9539.0, 7062.0]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node accuracy      time blank_control
0  1617.0      0.9  117.022702      8.484448
1  2606.0      1.0  129.348448      8.558365
2   8131.0      0.9   91.209498      8.218650
3 13588.0      1.0   88.568535      8.241789
4   9536.0      0.9   87.368412      8.275271
```

我们可以看到，随着 **b** 值的减少，正确率略有下降（因为哈希签名矩阵变小了，两个集合被哈希到同一个桶中的概率减少），但变化不大（因为如果两个集合真的相似，很可能在前面就已经遇到相同的哈希签名向量）。但是，运行的时间和算法所需的空间（即哈希签名矩阵的大小）都显著减少。

4.1.3 控制 **b=1564**

最后研究控制行条数不变的情况下改变每一行的行数会对正确率和运行时间产生什么影响。

```
27 1 conduct(1564,2)

18/69 18770.0 0.000000

[18770 rows x 2 columns]
A
[9528.0, 7065.0, 9531.0, 9537.0, 7063.0, 2464.0, 9539.0, 9535.0, 9533.0, 9532.0]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node accuracy      time blank_control
0  1617.0      0.9  250.190089      8.406944
1  2606.0      1.0  299.238387      8.614090
2   8131.0      1.0  177.864454      8.141581
3 13588.0      1.0  178.776733      8.256538
4   9536.0      1.0  177.061415      8.048621
```

```
28 1 conduct(1564,3)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[9537.0, 9528.0, 9531.0, 7065.0, 2464.0, 7063.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	192.015156	8.571047
1	2606.0	1.0	196.653999	8.639767
2	8131.0	1.0	173.982004	8.171142
3	13588.0	0.8	174.203591	8.151588
4	9536.0	1.0	177.575602	8.261116

```
29 1 conduct(1564,4)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[9528.0, 9531.0, 9537.0, 7065.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	220.769260	9.667596
1	2606.0	1.0	204.425007	9.881895
2	8131.0	1.0	197.905669	11.958325
3	13588.0	0.7	198.239826	9.435878
4	9536.0	0.8	197.023743	9.475883

```
In 24 1 conduct(1564,8)
```

```
18769 18770.0 0.000000
```

```
[18770 rows x 2 columns]
```

```
A
```

```
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
B
```

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	188.660291	9.565585
1	2606.0	0.0	174.105192	8.662297
2	8131.0	0.5	174.758080	8.269868
3	13588.0	0.3	174.992833	8.198900
4	9536.0	0.0	172.899221	8.289413

从结果来看，四个实验中的运行时间貌似变化不大。随着 r 值的增加，每次分桶需要比较的哈希签名向量维数增加，但进入候选集的节点变少，减少了 `similarest_ten()` 函数中的循环次数。但随着 r 越来越大，正确率下降幅度很大，空间消耗也越来越大。

4.1.4 减少行条数量

但是，即使正确率得到了保证，如果查询的时间反而远远大于暴力搜索的时间，那这个算法就没有什么意义了。注意到课件上只要两个集合的 Jaccard 相似度达到 0.8，哪怕在行条数为 20，每个行条有 5 行的情况下，这两个集合被哈希到同一个桶中的概率都大于 99%。所以，我决定把行条数缩小到“几十”这个数量级，再做一次实验。

```
In [10]: ► conduct(25,4)
```

```
4120    7063.0  0.142857
...
6285    11184.0  0.000000
6284    10230.0  0.000000
6283     9077.0  0.000000
6282     7847.0  0.000000
18769   18770.0  0.000000

[18770 rows x 2 columns]
A
[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node  accuracy    time  blank_control
0  1617.0      0.9  8.279038      24.699798
1  2606.0      0.1  7.205454      25.220060
2   8131.0      0.8  7.448853      24.266679
3 13588.0      0.2  7.548055      25.561765
4   9536.0      0.4  7.784551      24.088048
```

```
In [11]: ► conduct(50,2)
```

```
4120    7063.0  0.142857
...
6285    11184.0  0.000000
6284    10230.0  0.000000
6283     9077.0  0.000000
6282     7847.0  0.000000
18769   18770.0  0.000000

[18770 rows x 2 columns]
A
[9537.0, 2464.0, 7063.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node  accuracy    time  blank_control
0  1617.0      0.9 16.644711      25.944803
1  2606.0      1.0 16.729392      25.295576
2   8131.0      1.0 14.962687      23.593641
3 13588.0      0.2 14.656270      24.748156
4   9536.0      0.7 15.804278      24.647532
```

In [12]: ► conduct(25,2)

```
1617.0  7063.0  0.142857
...
6285  11184.0  0.000000
6284  10230.0  0.000000
6283   9077.0  0.000000
6282   7847.0  0.000000
18769 18770.0  0.000000

[18770 rows x 2 columns]
A
[9537.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node accuracy    time blank_control
0  1617.0      0.9  7.893611      24.370937
1   2606.0      0.9  7.696799      25.746443
2   8131.0      0.9  7.976640      24.992567
3  13588.0      0.2  7.360136      23.686425
4   9536.0      0.6  7.629885      23.443513
```

In [13]: ► conduct(75,2)

```
1617.0  7063.0  0.142857
...
6285  11184.0  0.000000
6284  10230.0  0.000000
6283   9077.0  0.000000
6282   7847.0  0.000000
18769 18770.0  0.000000

[18770 rows x 2 columns]
A
[9537.0, 2464.0, 7063.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node accuracy    time blank_control
0  1617.0      0.9 25.447755      24.202793
1   2606.0      1.0 24.208569      26.706301
2   8131.0      1.0 23.641568      23.672929
3  13588.0      0.2 22.382354      23.984605
4   9536.0      0.7 22.882921      25.620362
```

In [14]: ► conduct(75,3)

```
1617.0  7063.0  0.142857
...
6285  11184.0  0.000000
6284  10230.0  0.000000
6283   9077.0  0.000000
6282   7847.0  0.000000
18769 18770.0  0.000000

[18770 rows x 2 columns]
A
[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
   node accuracy    time blank_control
0  1617.0      0.9 25.170151      25.723858
1   2606.0      0.7 22.982913      25.065898
2   8131.0      0.9 22.520689      23.812155
3  13588.0      0.2 23.539508      25.031824
4   9536.0      0.4 22.207900      23.751745
```


In [15]: `conduct(25,3)`

```
18769 18770.0 0.000000
4120 7063.0 0.142857
...
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 1.0 7.858285 24.404414
1 2606.0 0.3 7.463274 25.377985
2 8131.0 0.9 7.332241 23.443647
3 13588.0 0.2 7.219323 23.559191
4 9536.0 0.4 7.181835 23.577977
```

In [16]: `conduct(50,3)`

```
4120 7063.0 0.142857
...
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 0.9 15.979109 24.237231
1 2606.0 0.5 14.823661 24.529499
2 8131.0 0.9 14.752694 23.284264
3 13588.0 0.2 14.594082 23.208277
4 9536.0 0.4 14.479938 23.327783
```

In [17]: `conduct(25,1)`

```
4120 7063.0 0.142857
...
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9537.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 7062.0, 9538.0, 9534.0]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 1.0 9.118827 24.009025
1 2606.0 1.0 13.090830 24.261395
2 8131.0 0.9 7.514032 22.962595
3 13588.0 0.5 7.316690 22.992976
4 9536.0 0.7 7.484177 23.368083
```

```
In [29]: conduct(50,4)
4120 7063.0 0.142857
...
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 0.9 16.089266 24.275946
1 2606.0 0.1 14.768803 25.180279
2 8131.0 0.8 14.975400 23.954664
3 13588.0 0.2 14.880749 23.968483
4 9536.0 0.4 15.059273 23.705806
```

我们可以发现，只要行条数小于 75，查询时间都小于暴力搜索的时间，对于大部分节点来说查询的正确率也都能得到保证。

但是，节点 2606 和 13588 的正确率无论在何种情况下都低于其他节点。以节点 2606 的邻居集合为例，所有集合中和 2606 的邻居集合 Jaccard 相似度最高的也只有 0.34，这个数值与其他集合与 2606 的邻居集合被分到同一个桶中的概率相等。所以，其它集合与 2606 的邻居集合被哈希到同一个桶中的概率较低，导致进入备选集的节点数量少于十个，最终导致这个节点的准确率很低。13588 的邻居集合也是同理。

```
In [29]: conduct(50,4)
[ 319. 1218. 1001. 14.]
volunteer
[2592, 1212, 2590]
Jaccard
node jaccard
0 2592.0 0.305195
2 2590.0 0.151042
1 1212.0 0.097778
Jaccard
node jaccard
3094 5842.0 0.343137
3658 4121.0 0.320755
3092 5838.0 0.317757
2723 2592.0 0.305195
2248 586.0 0.286885
...
7839 7778.0 0.000000
7840 1388.0 0.000000
7841 1390.0 0.000000
7842 821.0 0.000000
```

4.2 多探寻局部敏感哈希的运行结果

接着我们把普通 LSH 和多探寻 LSH 的正确率和运行时间作个比较,看看在空间消耗相等的情况下多探寻 LSH 是否能得到更好的结果。当然,多探寻 LSH 的 `conduct()` 函数要略作改动,因为它多了一个参数 `bound`。

```
def conduct(b,r,bound):
    global test_node
    global Verte
    df =
pd.DataFrame(columns=['node','accuracy','time','blank_control'],dtype
=float)
    for i in range (5):
        print("-----")
        print("i")
        print(i)
        t_a=time.perf_counter()
        A = LSH(test_node[i],b,r,bound)
        t_b=time.perf_counter()
        Verte_without_i = list(Characteristic_Graph.nodes)
        if test_node[i] in Verte_without_i:
            Verte_without_i.remove(test_node[i])
        t_c=time.perf_counter()
        B = similarest_ten(test_node[i],Verte_without_i)
        t_d=time.perf_counter()
        accuracy = len(list(set(A) & set(B)))/len(B)
        run_time = t_b-t_a
        compare_time = t_d-t_c
        df1 =
pd.DataFrame([[test_node[i],accuracy,run_time,compare_time]],columns=
['node','accuracy','time','blank_control'],dtype=float)
        df = pd.concat([df,df1], ignore_index=True)
        print("A")
        print(A)
        print("B")
        print(B)
    print(df)
```

一开始我们尝试着把 `bound` 放宽到 1 和 10,但跟普通的 LSH 相比优化作用不够显著。最后,我们把 `bound` 放宽到了 100,多探寻 LSH 和普通 LSH 的结果有了明显的不同。

- (3128, 4)

```
In 22 1 conduct(3128,4)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[9528.0, 9531.0, 9537.0, 7065.0, 9532.0, 9533.0, 9535.0, 9539.0, 7062.0, 9538.0]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	1.0	441.337044	9.604685
1	2606.0	1.0	417.922232	10.096952
2	8131.0	1.0	395.398334	9.472690
3	13588.0	0.7	392.771007	9.503786
4	9536.0	0.8	399.346558	9.684781

```
16 1 conduct(3128,4,100)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[9528.0, 9531.0, 9537.0, 7065.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	1169.977575	9.373333
1	2606.0	1.0	2178.565276	9.506556
2	8131.0	0.9	402.039683	9.267833
3	13588.0	0.7	388.446679	9.296995
4	9536.0	1.0	356.571701	8.411170

● (1564, 8)

```
In 24 1 conduct(1564,8)
```

18769 18770.0 0.000000

[18770 rows x 2 columns]

A

[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	188.660291	9.565585
1	2606.0	0.0	174.105192	8.662297
2	8131.0	0.5	174.758080	8.269868
3	13588.0	0.3	174.992833	8.198900
4	9536.0	0.0	172.899221	8.289413

```

17 1 conduct(1564,8,100)

18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
      node accuracy      time blank_control
0    1617.0      0.9 190.639449      8.471697
1    2606.0      0.7 176.446294      8.728456
2    8131.0      0.5 172.931645      8.104576
3   13588.0      0.3 173.030331      8.105900
4    9536.0      0.0 171.424233      8.522813

```

● (782, 16)

```

In 25 1 conduct(782,16)

18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
      node accuracy      time blank_control
0    1617.0      0.6 85.571515      8.208882
1    2606.0      0.0 86.795195      8.563412
2    8131.0      0.1 87.259601      8.065334
3   13588.0      0.0 86.316925      8.165431
4    9536.0      0.0 87.925898      8.193756

```

```

1 conduct(782,10,100)

18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
      node accuracy      time blank_control
0    1617.0      0.8 86.946427      8.353252
1    2606.0      0.0 87.046904      8.681575
2    8131.0      0.1 87.138357      8.625024
3   13588.0      0.0 88.270079      8.280407
4    9536.0      0.0 86.148890      8.258004

```

可以看到，当 **bound=100**，在哈希签名矩阵分割情况相同的条件下，三组实验的正确率都有了不同程度的提升。但是，多探寻 LSH 的运行时间相比普通 LSH 要稍微高一些，因为加入候选集合的节点更多了，`similarest_ten()` 中的遍历次数也会变多，但是时间的增加不是特别多。

让我们在降低行条数目以后再做一次实验：

- (25, 4)

In [10]: `conduct(25, 4, 1)`

```
18769  9531.0  0.000000
4120   7063.0  0.142857
...
6285  11184.0  0.000000
6284  10230.0  0.000000
6283   9077.0  0.000000
6282   7847.0  0.000000
18769  18770.0  0.000000
```

[18770 rows x 2 columns]

A

[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	7.864946	24.432336
1	2606.0	0.1	7.764069	27.052629
2	8131.0	0.8	7.926449	23.966281
3	13588.0	0.2	7.386726	23.789223
4	9536.0	0.4	7.373106	23.922398

In [19]: `conduct(25, 4, 10)`

```
18769  9531.0  0.000000
4120   7063.0  0.142857
...
6285  11184.0  0.000000
6284  10230.0  0.000000
6283   9077.0  0.000000
6282   7847.0  0.000000
18769  18770.0  0.000000
```

[18770 rows x 2 columns]

A

[9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1, -1]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	7.785095	25.218492
1	2606.0	0.1	7.768364	27.524722
2	8131.0	0.8	8.117975	24.529995
3	13588.0	0.2	7.519539	24.567248
4	9536.0	0.4	7.567986	24.468895

```
In [22]: conduct(25, 4, 100)
```

4120	7063.0	0.142857
...
6285	11184.0	0.000000
6284	10230.0	0.000000
6283	9077.0	0.000000
6282	7847.0	0.000000
18769	18770.0	0.000000

[18770 rows x 2 columns]

A

```
[7063.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, -1, -1, -1, -1]
```

B

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	7.938028	24.164110
1	2606.0	0.5	8.585558	25.170024
2	8131.0	0.8	7.584507	23.977910
3	13588.0	0.2	7.523212	23.757339
4	9536.0	0.5	7.465789	24.422271

● (50, 2)

```
In [11]: conduct(50, 2, 1)
```

4120	7063.0	0.142857
...
6285	11184.0	0.000000
6284	10230.0	0.000000
6283	9077.0	0.000000
6282	7847.0	0.000000
18769	18770.0	0.000000

[18770 rows x 2 columns]

A

```
[9537.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, -1]
```

B

```
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
```

	node	accuracy	time	blank_control
0	1617.0	0.9	16.605773	25.752200
1	2606.0	1.0	17.195903	26.189246
2	8131.0	1.0	14.969602	24.073957
3	13588.0	0.2	14.735047	23.736384
4	9536.0	0.7	15.816651	25.488575

```
In [20]: ► conduct(50, 2, 10)
```

4120	7063.0	0.142857
...
6285	11184.0	0.000000
6284	10230.0	0.000000
6283	9077.0	0.000000
6282	7847.0	0.000000
18769	18770.0	0.000000

[18770 rows x 2 columns]

A

[9537.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, 927.0]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	17.327014	25.066458
1	2606.0	1.0	21.115796	26.859987
2	8131.0	1.0	15.416627	24.410228
3	13588.0	0.2	14.996841	24.545109
4	9536.0	0.7	15.827213	26.242661

```
In [23]: ► conduct(50, 2, 100)
```

4120	7063.0	0.142857
...
6285	11184.0	0.000000
6284	10230.0	0.000000
6283	9077.0	0.000000
6282	7847.0	0.000000
18769	18770.0	0.000000

[18770 rows x 2 columns]

A

[9537.0, 9531.0, 9528.0, 2464.0, 7063.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0]

B

[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]

	node	accuracy	time	blank_control
0	1617.0	0.9	52.370322	24.443750
1	2606.0	1.0	88.624899	25.053032
2	8131.0	0.9	15.734835	24.006678
3	13588.0	0.4	15.942169	24.067297
4	9536.0	0.9	17.764496	24.103823

- (25, 2)

In [12]: ► conduct(25, 2, 1)

```
18769 18770.0 0.000000
4120 7063.0 0.142857
... ..
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9537.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, -1]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 0.9 7.950032 24.593995
1 2606.0 0.9 7.852103 26.903977
2 8131.0 0.9 7.774647 25.310109
3 13588.0 0.2 7.411748 23.893572
4 9536.0 0.7 8.078290 24.111703
```

In [21]: ► conduct(25, 2, 10)

```
18769 18770.0 0.000000
4120 7063.0 0.142857
... ..
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9537.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0, 9538.0, 9534.0, 927.0]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 1.0 8.497216 24.680836
1 2606.0 0.9 8.678280 25.458644
2 8131.0 0.9 7.638705 23.796086
3 13588.0 0.2 7.390524 23.793850
4 9536.0 0.7 7.478529 24.246504
```

In [24]: ► conduct(25, 2, 100)

```
18769 18770.0 0.000000
4120 7063.0 0.142857
... ..
6285 11184.0 0.000000
6284 10230.0 0.000000
6283 9077.0 0.000000
6282 7847.0 0.000000
18769 18770.0 0.000000

[18770 rows x 2 columns]
A
[9537.0, 9531.0, 9528.0, 2464.0, 7063.0, 9532.0, 9535.0, 9539.0, 9533.0, 9538.0]
B
[7065.0, 9537.0, 9528.0, 9531.0, 7063.0, 2464.0, 9532.0, 9533.0, 9535.0, 9539.0]
node accuracy time blank_control
0 1617.0 0.9 13.831750 24.494189
1 2606.0 1.0 28.886626 25.052404
2 8131.0 0.9 7.848143 24.134781
3 13588.0 0.4 7.852450 24.273630
4 9536.0 0.9 8.087313 24.578528
```

我们发现，多探寻局部敏感哈希在行条数比较小的时候也能在时间成本没有太大变化的前提下提升正确率。值得一提的是，由于哈希签名向量的长度变短了，我们在放宽的 `bound` 比较小的情况下就能看到正确率的提升。比如，对节点 9536 来说，在行条数为 25，每个行条有 2 行的前提下，只要把 `bound` 从 0 调到 1，我们就能把正确率从 0.6 提升到 0.7。

五、 结论（对使用的方法可能存在的不足进行分析，以及未来可能的研究方向进行讨论）

5.1 结论

- 实践表明，行条数 `b` 对运行时间的影响最大，运行时间会随着 `b` 的增加显著上升；
- `b` 对正确率的影响不大，所以为了保证正确率的情况下缩短查询时间，我们应该把 `b` 调整到 75 以下；
- 每个行条的行数 `r` 对正确率的影响很大，正确率会随着 `r` 的上升急剧下降，但 `r` 的变化对运行时间影响不大；
- 所以，可以适当控制 `b` 的大小并不要把 `r` 设置得太大，就能在相对较短的时间内得到正确率较高的搜索结果，这同时还减少了算法的空间开销；
- 如果使用多探寻 LSH，在 `b` 和 `r` 相同的情况下，还可以用略微增加一点运行时间的代价在 `r` 比较大的时候得到更高的正确率，这在哈希签名举着较大而不得不控制 `b` 的大小的时候非常有用。

5.2 可能存在的不足

- 本次实验中，最大的遗憾就是 LSH 算法把大把的时间用在了分桶上，而分桶过程中还有着不少的重复计算；
- 此外，我们没有对哈希函数本身进行优化，使得相似的集合被哈希到同一个桶中的概率更大。

5.3 未来可能的研究方向

- 可以对分桶的过程进行优化，从而减少循环次数、避免重复计算；
- 目前有人已经提出了用 Hadamard 变换的方法来优化哈希函数¹，但这种方法只能用于集合数量为偶数的情况，未来还可以研究对所有情况都适用的哈希函数优化方案。

¹ Anirban Dasgupta.Ravi Kumar.Tamás Sarlós.Fast Locality-Sensitive Hashing