南京理工大学

算法课程设计

姓名：高宏艳

学号：919106840202

指导老师：周沧琦

目录

1 数独问题

1.1 问题描述

数独（Sudoku）是一款数字逻辑游戏，玩家需要根据 9X9 盘面上的已知数字（如图 1），推算出所有剩余空格的数字，并且满足每一行、每一列、每一个粗线宫内的数字均含 1-9，并且不重复。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

图 1 数独示意图

本次实验需要解决的基本问题有：

- 基于回溯算法解决数独问题
- 探讨不可解的情况
- 探讨多解的情况

1.2 算法思路

1.2.1 回溯算法

- **基本思路**：利用结构体 VAC 储存数独问题当中所有空缺位置的信息（包含空缺位置的行号、列号、宫格号以及该空格内的可行解列表），并且考虑求解的时效性，将空缺位

置按照其可行解列表的长度升序排序。按顺序遍历空缺位置进行填充，并检查当前空缺位置填充的合法性，若合法则进行下一个空缺位置的填充，若不合法则进行回溯，若刚好遍历到最后一个空缺位置，则进行一次答案输出，并对输出答案次数进行计数。当所有搜索结束，若输出答案计数为零，则该数独无解；否则，输出该数独的解的个数。

图 2 九宫格序号的划分

1.2.2 直观遍历求解

- **基本思路**：对于输入的数独题目不进行任何预处理，从第 1 行第 1 列进逐列逐行的遍历，若为空格，则挨个尝试 1-9，isSafe 函数判断是否填入的是合法值，若成功填入则进行下一个位置的试探，直到达到最后一行最后一列或是得到正确的解停止。

1.3 算法设计

1.3.1 回溯法设计

步骤：

1) 初始化输入的数独地图，将所有空位置信息（行号、列号、宫格号以及初始地图生成对应空位置的可行解列表）存储到空位置列表中，按照每个空位置的可行解列表长度升序排列，同时初始化计数器为零
2) 遍历空位置列表，对于当前空位置，根据它的可行解列表，抽取列表表头的可行解进行填入，进行 1）
3) 如果步骤 2) 中这个空位置的所有未论证的解为空，则说明上一次的随机填入有误，进行步骤 4）
4) 退到上一个空位置，将上一次填入的解从未论证的解中删除，更新数独状态，进行步骤 2）
5) 若发现所有空位置已经填完，则说明完成数独解，输出答案，并且计数器加一，进行步骤 4），此步骤是为了解决多解的问题

流程图：

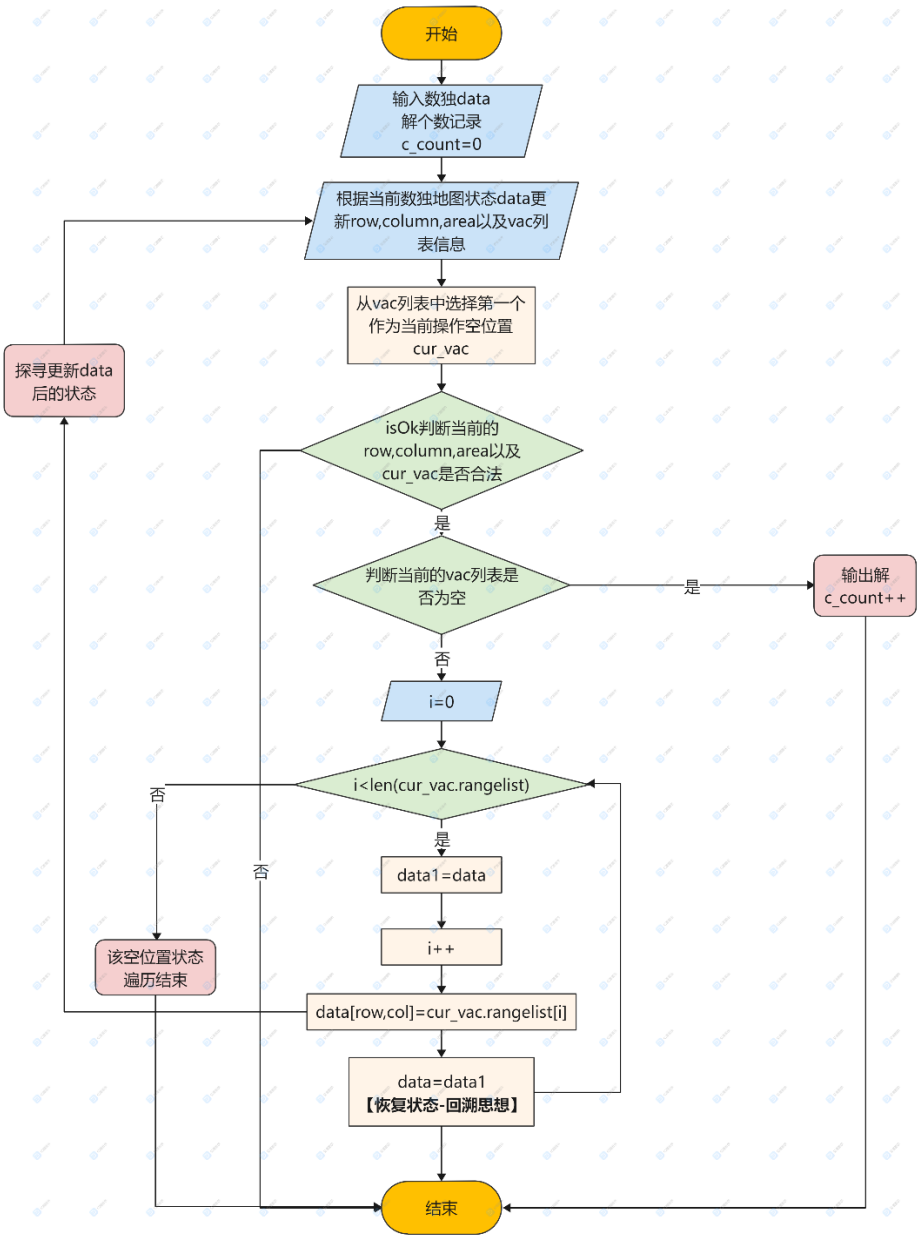

图 3 回溯法求解数独问题流程图

回溯函数伪代码:

```
global c_count=0

def solve_1(data):

    row,column,area=_init(data)# 初始化行、列以及九宫格的状态
    vac=_initVac(data,row,column,area)# 初始化空位置的状态
    # 如果不通过
    if isOk(row,column,area,vac)==False:
        return 0
    # 如果空缺位置已经全部填满
```

```
    if vac==[]:
        # 输出求解结果
        print(data)
        c_count+=1 # 计数器+1
        return 0
    # 否则继续搜索
    else:
        cur_vac=vac[0] #当前的空缺位置
        if len(cur_vac.rangelist)==1:
            data[cur_vac.row,cur_vac.col]=cur_vac.rangelist[0]
                        return solve_1(data)
        else if len(cur_vac.rangelist)>1:
            data1=copy.deepcopy(data)
            for i in cur_vac.rangelist:
                data[cur_vac.row,cur_vac.col]=i
                solve_1(data)
                data=data1 # 回溯
            return 0
```

1.3.2 直观遍历设计

步骤：

1) 将输入的数独题目保存为 9x9 矩阵形式，初始化计数器 count=0
2) 创建一个函数 isSafe，检查数独是否为有效数独（即每一行、每一列、每一个九宫格内没有重复的数字），如果数独不合法则返回 False 反之返回 True
3) 创建一个递归函数 Solve()，改函数接受一个数独矩阵以及当前行列索引：如果当前行列出于矩阵末尾，即 i=N-1 和 j=N,则检查矩阵是否合法，若合法则打印当前的数独解，并且 count++;另一种基本情况是，如果列的值为 N，即 j=N,则遍历下一行，i++且 j=0;若未指定到当前索引，则将元素从填充到 9，并使用下一个元素的索引重复出现的 9 种情况；
   若分配到了当前索引，则使用下一个元素的索引调用递归函数

流程图：

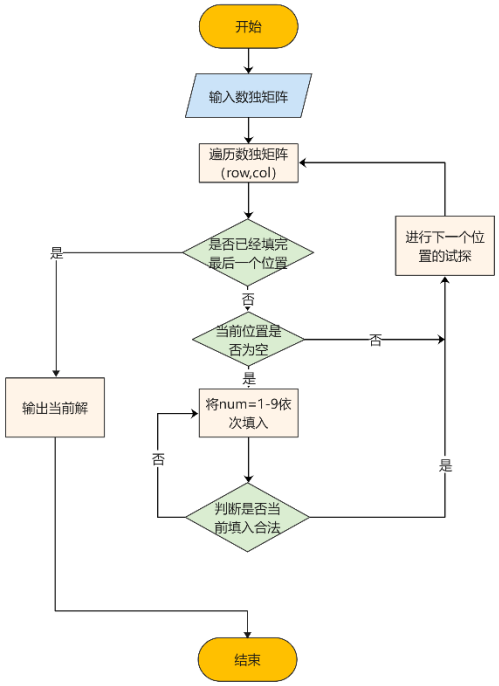

图 4 直观遍历求解数独问题流程图

伪代码:

```
count=0
def solve_00(data,row,col):
    """
    输入棋盘，以及行列信息
    :param data: 棋盘
    :param row: 行
    :param col: 列
    :return: 是否有解
    """
    global count
    if(row==8 and col==9 ):
        # 到达边界，说明求解成功
        count+=1
        printData(data)  # 输出求解结果
        return 0

    if col==9:
        row+=1
        col=0

    if data[row][col]>0:
        # 已经有数字
```

```
23.         return solve_00(data,row,col+1)  #递归调用
24.     for num in range(1,10,1):
25.         # 1-9 遍历
26.         if isSafe(data,row,col,num):
27.             data[row][col]=num
28.
29.             if solve_00(data,row,col+1):
30.                 return 0
31.         # 还原这个位置的的值
32.         data[row][col]=0
33.
34.     return 0
```

1.4 算法实现细节

1.4.1 语言

Python

1.4.2 环境

开发环境：Pycharm+python3.8

表 1 数独问题所用到的包

包名	作用
copy	复制
time	计算程序运行时间
numpy	科学计算
threading	对于特定函数限时运行，检验算法运行效率

1.4.3 关键步骤的实现细节

1. 数据结构：
   1) 空缺位置 VAC

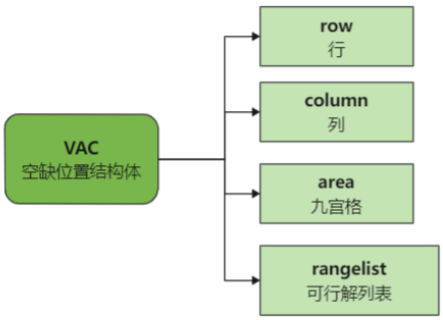

图 5 VAC 结构体

2) 数独地图 data=np.array([]).reshape(9,9),存储最初输入的数独题目
3) 数独行状态 row=[[],[],[],[],[],[],[],[],[]],存储每一行出现的数字
4) 数独列状态 column=[[],[],[],[],[],[],[],[],[]]存储每一列出现的数字
5) 数独九宫格状态 area=[[],[],[],[],[],[],[],[],[]]存储每一个序号内的九宫格出现的数字

2. 主要函数:

1) 初始化行、列、九宫格状态**_init(data):**
将当前数独地图的信息存储到 row,column,area 的对应下标的列表中
返回 row,column,area 列表

2) 初始化空缺位置信息**_initVac(data,row,col,area)**
根据数独地图信息来获取空缺位置的行列编号以及九宫格对应编号，对于{0-9}与空缺位置所在行、列、九宫格位置列表数据求差集得到对应空缺位置的可行解列表。返回所有空缺位置的 vac 列表。

3) 回溯法求解 **solve_1(data)**
求解过程见图 3

4) 判断当前数独状态是否为合法过程解或者终解 **isOk(row,column,area,vac)**
如果当前数独棋盘 data 中有某一行或者某一列或者某一个九宫格内数据冲突，或者存在某空缺位置可行解列表为空，则返回 False，否则返回 True.

1.5 运行结果

1.5.1 单解的情况

1.5.1.1 回溯法

1. 单解测试一（耗时 1.049s）
输入

```
[[0 0 5 3 0 0 0 0 0]
 [8 0 0 0 0 0 0 2 0]
 [0 7 0 0 1 0 5 0 0]
 [4 0 0 0 0 5 3 0 0]
 [0 1 0 0 7 0 0 0 6]
 [0 0 3 2 0 0 0 8 0]
 [0 6 0 5 0 0 0 0 9]
 [0 0 4 0 0 0 0 3 0]
 [0 0 0 0 0 9 7 0 0]]
```

输出

```
[[1 4 5 3 2 7 6 9 8]
 [8 3 9 6 5 4 1 2 7]
 [6 7 2 9 1 8 5 4 3]
 [4 9 6 1 8 5 3 7 2]
 [2 1 8 4 7 3 9 5 6]
 [7 5 3 2 9 6 4 8 1]
 [3 6 7 5 4 2 8 1 9]
 [9 8 4 7 6 1 2 3 5]
 [5 2 1 8 3 9 7 6 4]]
```

2. 单解测试二（耗时：0.062s）

输入

```
[[0 0 0 5 8 7 0 0 0]
 [0 6 5 4 0 3 9 7 0]
 [0 4 0 0 0 0 0 5 0]
 [5 9 0 0 6 0 0 4 2]
 [4 0 0 3 0 2 0 0 9]
 [2 8 0 0 4 0 0 6 5]
 [0 2 0 0 0 0 0 9 0]
 [0 5 8 2 0 6 4 1 0]
 [0 0 0 1 9 5 0 0 0]]
```

输出

```
[[1 3 9 5 8 7 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [7 4 2 6 1 9 8 5 3]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [3 2 1 8 7 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
耗时： 0.062497854232788086 s
```

1.5.1.2 直观遍历求解

1. 单解测试（耗时 0.372s）
   输入：

```
data = [[0,0,5,3,0,0,0,0,0],
    [8,0,0,0,0,0,0,2,0],
    [0,7,0,0,1,0,5,0,0],
    [4,0,0,0,0,5,3,0,0],
    [0,1,0,0,7,0,0,0,6],
    [0,0,3,2,0,0,0,8,0],
    [0,6,0,5,0,0,0,0,9],
    [0,0,4,0,0,0,0,3,0],
    [0,0,0,0,0,9,7,0,0]]
```

   输出：

```
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4
耗时： 0.37202978134155273
```

1.5.2 无解的情况

1.5.2.1 回溯法

1. 无解测试一（耗时 0.000s）
   输入数独本身不满足数独游戏存在解的条件（即最初棋盘中存在冲突）
   输入

```
[[0 0 0 5 8 7 0 0 0]
 [0 6 5 4 0 3 9 7 0]
 [0 4 0 0 0 0 0 5 0]
 [5 9 0 0 6 0 0 4 2]
 [4 8 0 3 0 2 0 0 9]
 [2 8 0 0 4 0 0 6 5]
 [0 2 0 0 0 0 0 9 0]
 [0 5 8 2 0 6 4 1 0]
 [0 0 0 1 9 5 0 0 0]]
```

输出

```
~~~~~~无解~~~~~~
耗时： 0.0 s
~~~~~~无解~~~~~~
```

2. 无解测试二（耗时 0.016s）
   输入数独无法得到可行解
   输入

```
[[0 0 0 5 8 7 0 0 0]
 [0 6 5 4 0 3 9 7 0]
 [0 4 0 0 0 0 0 5 0]
 [5 9 0 0 6 0 0 4 2]
 [4 0 0 3 0 2 0 0 9]
 [2 8 0 0 4 0 0 6 5]
 [6 2 0 0 0 0 0 9 0]
 [0 5 8 2 0 6 4 1 0]
 [0 0 0 1 9 5 0 0 0]]
```

输出

```
耗时： 0.01564645767211914 s
~~~~~~无解~~~~~~
```

1.5.2.2 直观遍历求解

1. 无解测试一：
   输入：

```
data = [[0,0,0,5,8,7,0,0,0],
    [0,6,5,4,0,3,9,7,0],
    [0,4,0,0,0,0,0,5,0],
    [5,9,0,0,6,0,0,4,2],
    [4,8,0,3,0,2,0,0,9],
    [2,8,0,0,4,0,0,6,5],
    [0,2,0,0,0,0,0,9,0],
    [0,5,8,2,0,6,4,1,0],
    [0,0,0,1,9,5,0,0,0]]
```

输出：

```
无解
耗时：  0.0150170326232910 16
```

1.5.3 多解的情况

1.5.3.1 回溯法

1. 多解测试一（耗时 0.171s）
   输入

```
[[0 0 0 0 0 0 0 0 0]
 [0 6 5 4 0 3 9 7 0]
 [0 4 0 0 0 0 0 5 0]
 [5 9 0 0 6 0 0 4 2]
 [4 0 0 3 0 2 0 0 9]
 [2 8 0 0 4 0 0 6 5]
 [0 2 0 0 0 0 0 9 0]
 [0 5 8 2 0 6 4 1 0]
 [0 0 0 1 9 5 0 0 0]]
```

   输出（14 个解）

```
******************** 求解结果 ********************
[[1 7 2 5 8 9 6 3 4]
 [8 6 5 4 2 3 9 7 1]
 [3 4 9 6 1 7 2 5 8]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [6 2 1 8 7 4 5 9 3]
 [9 5 8 2 3 6 4 1 7]
 [7 3 4 1 9 5 8 2 6]]
******************** 求解结果 ********************
[[1 7 2 5 8 9 6 3 4]
 [8 6 5 4 2 3 9 7 1]
 [3 4 9 6 1 7 2 5 8]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [7 2 1 8 3 4 5 9 6]
 [9 5 8 2 7 6 4 1 3]
 [6 3 4 1 9 5 8 2 7]]
耗时： 0.17186927795410156 s
~~~~~~~~~~~~~~~~~~有 14 个解~~~~~~~~~~~~~~~~~~
```

```
[[1 3 9 5 7 8 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [7 4 2 6 1 9 8 5 3]
 [5 9 3 8 6 7 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [3 2 1 7 8 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
******************** 求解结果 ********************
[[1 3 9 5 7 8 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [7 4 2 6 1 9 8 5 3]
 [5 9 7 8 6 1 3 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 3 9 4 7 1 6 5]
 [3 2 1 7 8 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
******************** 求解结果 ********************
[[1 3 9 5 8 7 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [7 4 2 6 1 9 8 5 3]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [3 2 1 8 7 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
```

```
[[7 3 9 5 1 8 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [1 4 2 6 7 9 8 5 3]
 [5 9 3 8 6 7 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [3 2 1 7 8 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
******************** 求解结果 ********************
[[7 3 9 5 1 8 6 2 4]
 [8 6 5 4 2 3 9 7 1]
 [1 4 2 6 7 9 8 5 3]
 [5 9 7 8 6 1 3 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 3 9 4 7 1 6 5]
 [3 2 1 7 8 4 5 9 6]
 [9 5 8 2 3 6 4 1 7]
 [6 7 4 1 9 5 2 3 8]]
******************** 求解结果 ********************
[[7 1 2 5 8 9 6 3 4]
 [8 6 5 4 2 3 9 7 1]
 [3 4 9 6 1 7 2 5 8]
 [5 9 1 7 6 8 3 4 2]
 [4 7 6 3 5 2 1 8 9]
 [2 8 3 9 4 1 7 6 5]
 [1 2 7 8 3 4 5 9 6]
 [9 5 8 2 7 6 4 1 3]
 [6 3 4 1 9 5 8 2 7]]
******************** 求解结果 ********************
```

```
[[8 7 2 5 1 9 6 3 4]
 [1 6 5 4 2 3 9 7 8]
 [3 4 9 6 7 8 2 5 1]
 [5 9 3 8 6 7 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [6 2 1 7 8 4 5 9 3]
 [9 5 8 2 3 6 4 1 7]
 [7 3 4 1 9 5 8 2 6]]
********** 求解结果 **********
[[8 7 2 5 1 9 6 3 4]
 [1 6 5 4 2 3 9 7 8]
 [3 4 9 6 7 8 2 5 1]
 [5 9 7 8 6 1 3 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 3 9 4 7 1 6 5]
 [6 2 1 7 8 4 5 9 3]
 [9 5 8 2 3 6 4 1 7]
 [7 3 4 1 9 5 8 2 6]]
********** 求解结果 **********
[[8 7 2 5 1 9 6 3 4]
 [1 6 5 4 2 3 9 7 8]
 [3 4 9 6 8 7 2 5 1]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [6 2 1 8 3 4 5 9 7]
 [9 5 8 2 7 6 4 1 3]
 [7 3 4 1 9 5 8 2 6]]
********** 求解结果 **********
```

```
[[8 7 2 5 1 9 6 3 4]
 [1 6 5 4 2 3 9 7 8]
 [3 4 9 6 8 7 2 5 1]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [6 2 1 8 7 4 5 9 3]
 [9 5 8 2 3 6 4 1 7]
 [7 3 4 1 9 5 8 2 6]]
********** 求解结果 **********
[[8 7 2 5 1 9 6 3 4]
 [1 6 5 4 2 3 9 7 8]
 [3 4 9 6 8 7 2 5 1]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [7 2 1 8 3 4 5 9 6]
 [9 5 8 2 7 6 4 1 3]
 [6 3 4 1 9 5 8 2 7]]
********** 求解结果 **********
[[1 7 2 5 8 9 6 3 4]
 [8 6 5 4 2 3 9 7 1]
 [3 4 9 6 1 7 2 5 8]
 [5 9 3 7 6 8 1 4 2]
 [4 1 6 3 5 2 7 8 9]
 [2 8 7 9 4 1 3 6 5]
 [6 2 1 8 3 4 5 9 7]
 [9 5 8 2 7 6 4 1 3]
 [7 3 4 1 9 5 8 2 6]]
********** 求解结果 **********
```

2. 多解测试二（耗时 35.343s）

输入：

```
[[0 0 5 3 0 0 0 0 0]
 [8 0 0 0 0 0 0 0 0]
 [0 7 0 0 1 0 0 0 0]
 [4 0 0 0 0 5 0 0 0]
 [0 1 0 0 7 0 0 0 6]
 [0 0 3 2 0 0 0 8 0]
 [0 6 0 5 0 0 0 0 9]
 [0 0 4 0 0 0 0 3 0]
 [0 0 0 0 0 9 7 0 0]]
```

输出：(1406 个解)

注意*由于版面限制，这里只展现最初的两个解以及最后的两个输出解，完整的实验验证可以运行”回溯法_ghy.py”中的 test_7()函数进行检验)

```
******************** 求解结果 ********************
[[6 4 5 3 2 7 1 9 8]
 [8 3 1 9 5 4 6 2 7]
 [9 7 2 6 1 8 3 5 4]
 [4 8 6 1 9 5 2 7 3]
 [2 1 9 8 7 3 5 4 6]
 [7 5 3 2 4 6 9 8 1]
 [3 6 7 5 8 2 4 1 9]
 [5 9 4 7 6 1 8 3 2]
 [1 2 8 4 3 9 7 6 5]]
******************** 求解结果 ********************
[[6 4 5 3 2 7 1 9 8]
 [8 3 1 9 5 6 2 4 7]
 [9 7 2 4 1 8 5 6 3]
 [4 8 6 1 9 5 3 7 2]
 [2 1 9 8 7 3 4 5 6]
 [7 5 3 2 6 4 9 8 1]
 [3 6 7 5 4 1 8 2 9]
 [1 9 4 7 8 2 6 3 5]
 [5 2 8 6 3 9 7 1 4]]
******************** 求解结果 ********************
[[6 4 5 3 2 7 1 9 8]
 [8 3 1 9 5 6 2 7 4]
 [9 7 2 4 1 8 5 6 3]
 [4 8 6 1 9 5 3 2 7]
 [8 3 6 7 5 2 4 9 1]
 [2 7 9 4 1 8 6 5 3]
 [4 8 7 9 6 5 3 1 2]
 [5 1 2 8 7 3 9 4 6]
 [6 9 3 2 4 1 5 8 7]
 [7 6 1 5 3 4 8 2 9]
 [9 2 4 6 8 7 1 3 5]
 [3 5 8 1 2 9 7 6 4]]
******************** 求解结果 ********************
[[1 4 5 3 9 6 2 7 8]
 [8 3 6 7 5 2 9 4 1]
 [2 7 9 4 1 8 6 5 3]
 [4 8 7 9 6 5 3 1 2]
 [5 1 2 8 7 3 4 9 6]
 [6 9 3 2 4 1 5 8 7]
 [7 6 1 5 3 4 8 2 9]
 [9 2 4 6 8 7 1 3 5]
 [3 5 8 1 2 9 7 6 4]]
******************** 求解结果 ********************
[[1 4 5 3 9 2 6 7 8]
 [8 3 6 4 5 7 2 9 1]
 [2 7 9 6 1 8 4 5 3]
 [4 8 7 9 6 5 3 1 2]
 [5 1 2 8 7 3 9 4 6]
 [6 9 3 2 4 1 5 8 7]
 [7 6 1 5 3 4 8 2 9]
 [9 2 4 7 8 6 1 3 5]
 [3 5 8 1 2 9 7 6 4]]
耗时： 35.34317064285278 s
~~~~~~~~~~~~~~~~~~有 1406 个解~~~~~~~~~~~~~~~~~~
```

3. 多解测试三：对于全空数独棋盘的求解

输入：

```
********** 原始数独 **********
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
```

输出：2s 内的求解个数

对于全空数独的求解，2005 年由 Bertram Felgenhauer 和 Frazer Jarvis 计算得出约有 6,670,903,752,021,072,936,960（6.67×10 的 21 次方）种组合，如果将重复（如数字交换、对称等）不计算，那么有 5,472,730,538 个组合。这对于本次实验所用到的计算机显然有点牵强，于是本文中对于该问题的求解改成了用限时计算求解个数来衡量算法效率，同时为了避免输入输出造成的额外的时间损耗，将源码中的输出部分暂时注释掉，最终在 python3.8+pycharm 的环境下,2s 内回溯法可以计算出的解的个数为 580.

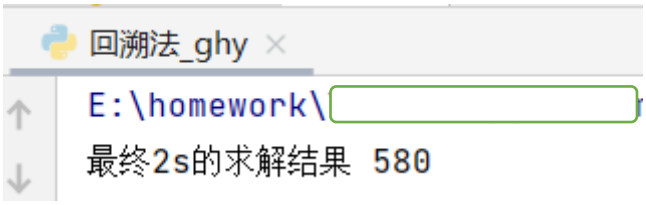

图 6 全 0 数独的限时求解结果

1.5.3.2 直观遍历求解

1. 多解测试一(耗时 0.417s):

   输入：

```
data = [[0,0,0,5,8,7,0,0,0],
        [0,6,5,4,0,3,9,7,0],
        [0,4,0,0,0,0,0,5,0],
        [5,9,0,0,6,0,0,4,2],
        [4,8,0,3,0,2,0,0,9],
        [2,8,0,0,4,0,0,6,5],
        [0,2,0,0,0,0,0,9,0],
        [0,5,8,2,0,6,4,1,0],
        [0,0,0,1,9,5,0,0,0]]
```

   输出：(14 组解，版面原因，这里只列出部分解)

```
耗时： 0.41674065589904785
```

求解结果
1 3 9 5 7 8 6 2 4
8 6 5 4 2 3 9 7 1
7 4 2 6 1 9 8 5 3
5 9 3 8 6 7 1 4 2
4 1 6 3 5 2 7 8 9
2 8 7 9 4 1 3 6 5
3 2 1 7 8 4 5 9 6
9 5 8 2 3 6 4 1 7
6 7 4 1 9 5 2 3 8
求解结果
1 3 9 5 7 8 6 2 4
8 6 5 4 2 3 9 7 1
7 4 2 6 1 9 8 5 3
5 9 7 8 6 1 3 4 2
4 1 6 3 5 2 7 8 9
2 8 3 9 4 7 1 6 5
3 2 1 7 8 4 5 9 6
9 5 8 2 3 6 4 1 7
6 7 4 1 9 5 2 3 8
求解结果
1 3 9 5 8 7 6 2 4
8 6 5 4 2 3 9 7 1

6 2 1 8 7 4 5 9 3
9 5 8 2 3 6 4 1 7
7 3 4 1 9 5 8 2 6
求解结果
1 7 2 5 8 9 6 3 4
8 6 5 4 2 3 9 7 1
3 4 9 6 1 7 2 5 8
5 9 3 7 6 8 1 4 2
4 1 6 3 5 2 7 8 9
2 8 7 9 4 1 3 6 5
7 2 1 8 3 4 5 9 6
9 5 8 2 7 6 4 1 3
6 3 4 1 9 5 8 2 7
求解结果
7 1 2 5 8 9 6 3 4
8 6 5 4 2 3 9 7 1
3 4 9 6 1 7 2 5 8
5 9 1 7 6 8 3 4 2
4 7 6 3 5 2 1 8 9
2 8 3 9 4 1 7 6 5
1 2 7 8 3 4 5 9 6
9 5 8 2 7 6 4 1 3
6 3 4 1 9 5 8 2 7
求解结果
7 3 9 5 1 8 6 2 4
8 6 5 4 2 3 9 7 1
1 4 2 6 7 9 8 5 3
5 9 3 8 6 7 1 4 2
4 1 6 3 5 2 7 8 9
2 8 7 9 4 1 3 6 5

求解结果
8 7 2 5 1 9 6 3 4
1 6 5 4 2 3 9 7 8
3 4 9 6 8 7 2 5 1
5 9 3 7 6 8 1 4 2
4 1 6 3 5 2 7 8 9
2 8 7 9 4 1 3 6 5
6 2 1 8 7 4 5 9 3
9 5 8 2 3 6 4 1 7
7 3 4 1 9 5 8 2 6
求解结果
8 7 2 5 1 9 6 3 4
1 6 5 4 2 3 9 7 8
3 4 9 6 8 7 2 5 1
5 9 3 7 6 8 1 4 2
4 1 6 3 5 2 7 8 9
2 8 7 9 4 1 3 6 5
7 2 1 8 3 4 5 9 6
9 5 8 2 7 6 4 1 3
6 3 4 1 9 5 8 2 7

1.6 问题探讨

1.6.1 算法复杂性分析

- **回溯法**:

  时间复杂度为 $O(\sum_{i=1}^{n} e_i)$，其中 n 为空缺位置个数，e_i为每一个空缺位置的可行解 list 内元素个数。对于每一个空缺位置，有e_i个可能的选项，因此时间复杂度的上界为$\sum_{i=1}^{n} e_i$。具体的时间复杂度需要根据输入的数独题目的复杂度、空缺位置个数来决定。然而具体函数在实现过程中，首先会对于每一个空缺位置的状态、可行解进行判断与预处理，这些早期的剪枝操作使得实际算法在处理多解数独以及一些无解的数独的时候，效率显著提高。

- **直观遍历求解**:

  同回溯法，时间复杂度为 O(9^(nxn))，对于每一个空缺位置，有 9 个可能的选项，因此时间复杂度的上界为 9^(nxn)。具体的时间复杂度需要根据输入的数独题目的复杂度、空缺位置个数来决定。

- **对比**：直观遍历求解在大多数单解情况下展现出的效率低于回溯法，在多解以及无解的任务下下时间消耗明显远远大于回溯法。由此可见回溯法在求解数独问题上具有更好的

时间效率。

2 社团发现问题

2.1 问题描述

在社交网络中，有的用户之间的连接较为紧密，有的用户之间的连接关系较为稀疏，在这样的的网络中，连接较为紧密的部分可以被看成一个社区，其内部的节点之间有较为紧密的连接，而在两个社区间则相对连接较为稀疏，这便称为社团结构。

社区划分算法可以有效地对于未知标签的网络进行社区划分，实际应用中可以方便互联网公司进行用户画像等工作。

Louvain 算法是一种较为简单的社区发现算法，其核心思想是一种基于模块度最大化的贪心策略。

该算法的优点在于速度快，可以在较短时间内实现大规模网络以不同粒度的社区划分，并且无需指定社区的数量，当模块度不再增益时候，迭代便自动停止。

·本次实验的目标：给定两个数据集，输出依据 Louvain 算法得出的社区划分结果。

- 对于海豚数据集进行社区划分，输出模块度取值
- 对于 Emial 数据集进行社区划分，输出模块度取值和 Accuracy

2.2 算法思路

2.2.1 基本概念

1) 无向图

无向图是一种简单的图模型，在这种图模型中，边仅仅是两个顶点之间的连接。

2) 模块度

模块度又称模块化度量值，是常用的一种衡量网络社区结构强度的方法，最早由 Mark NewMan 提出[3]。模块度的定义为：

$$Q = \frac{1}{2m}\sum_{i,j}[A_{ij} - \frac{k_i k_j}{2m}]\delta(c_i, c_j)$$

其中A_{ij}代表的是结点i和结点j之间的边权；如果$c_i = c_j$则函数$\delta(c_i, c_j) = 1$否则$\delta(c_i, c_j) = 0$；c_i表示的是结点i属于的社区；$m = \frac{1}{2}\sum_{i,j}A_{ij}$。

模块度值的大小主要取决于网络中结点的社区分配，即网络的社区划分情况，可以用来定量的衡量网络社区划分的质量，其值越接近 1，表示网络划分出的社区结构的强度越强，也就是划分质量越好。因此可以最大化模块度 Q 来获得最优网络社区划分。

3) 模块度增益

将一个单独的结点i移动到社区C中时，模块度增益可以由以下公式推算：

$$\Delta Q = [\frac{\sum_{in} + 2k_{i,in}}{2m} - (\frac{\sum_{tot} + 2k_i}{2m})^2] - [\frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 - (\frac{k_i}{2m})^2]$$

化简后可以写成：

$$\Delta Q = \frac{k_{i,in}}{m} - \frac{2k_i\sum_{tot}}{(2m)^2}$$

其中，$\sum_{tot}$是关联到社区C中的结点i连接的权重之和，k_i是关联到结点i的边的权重之和，$k_{i,in}$是从结点i到社区C中所有结点的边权之和。m是图中所有边的权重之和。

4) **聚类精确度**

聚类精确度（Accuracy,AC）用于比较获得的标签和数据提供的真实标签，AC 的计算公式如下：

$$AC = \frac{\sum_{i=1}^{n}\delta(s_i, map(r_i))}{n}$$

其中r_i和s_i分别表示数据x_i所对应的获得的标签和真实标签，n为数据的总个数，δ表示的指示函数如下：

$$\delta(x,y) = \begin{cases} 1, if\ x = y \\ 0\ oterwise \end{cases}$$

而式子中的map表示最佳分类的重现分配，以保证统计的正确性。

5) **ARI**：

ARI 又称调兰德指数，范围在[-1,1]，值越大越好，反应两种划分的重叠程度，计算公式为：

$$ARI = \frac{RI - E[RI]}{max(RI) - E[RI]}$$

其中 RI 的计算公式如下：

$$RI = \frac{a+b}{(C_n^2)}$$

n 表示实例总数，用 L 表示实际的类别划分，K 表示聚类结果，定义 a 为在 L 中被划分且在 K 中被划分为同一社区的实例对数量，定义 b 为在 L 中被划分为不同类别，在 K 中被划分为不同类别的实例对数量。

6) **互信息 MI**：

互信息是用来评价相同数据的两个标签之间的相似性度量，其公式如：

$$MI(U,V) = \sum_{i=1}^{|U|}\sum_{j=1}^{|V|}\frac{|U_i \cap V_j|}{N} log\frac{N|U_i \cap V_j|}{|U_i||V_j|}$$

其中，$|U_i|$是聚类簇U_i中的样本数；$|V_i|$是聚类簇V_j中的样本数

7) **FMI**：

FMI 是对于聚类结果和真实值计算得到的召回率和精确率进行几何平均的结果，取值范围为[0,1],越接 1 越好。

2.2.2 算法原理

Louvain 算法是一种基于图数据的社区发现算法，主要针对的是无向图。Louvain 算法的优化目标为最大化整个图数据的模块度。

Louvain 算法的是一种贪心算法，其核心在于两部迭代设计。

最开始，每个原始的结点都看成一个独立的社区，社区内的连边权重为 0.

- **步骤 1：**
  算法扫描图中所有的结点，针对每个结点遍历该结点的所有邻居结点，衡量将该结点加入其邻居结点所带来的模块度收益，并选择对应最大收益的邻居结点，加入其所在的社区。这一过程重复进行直到每一个结点的社区归属都不再发生变化。
- **步骤 2：**
  对步骤 1 中形成的社区进行折叠，把每一个社区折叠成一个单点，分别计算这些新生成的“社区点”之间的连边权重，以及社区内部所有点的连边权重之和。

一旦算法的步骤 2 完成，就可以将算法的步骤 1 重新应用到最新加权的网络并进行迭代，直到没有更多结点社区从属的更改并且达到模块度最大化。

值得注意的是，为了降低时间复杂性，可以引入一个人工设定的阈值，当模块度增益没有超过该阈值时即可终止算法。

2.3 算法设计

2.3.1 步骤与流程图

步骤:

1) 图中每个结点都视为一个社区
2) 对于每个结点 i，依次尝试将结点 i 加入到结点 i 对应的邻居结点的社区，并计算加入前与加入后模块度增量ΔQ，记录ΔQ 最大的结点所在的社区编号，如果 max(ΔQ)>0，则把结点 i 加入到该邻居结点所在的社区，否则结点 i 仍在原始社区
3) 重述上述过程，直到所有的结点不再发生移动
4) 对上图进行压缩，将所有在同一个社区的结点压缩成一个新结点，社区内结点之间的权重转化为新的结点的环的权重，社区间的边权重转换为新结点之间的边权重
5) 重复过程 1)和 2)，直到整个社区的模块度不再发生变化。

流程图：

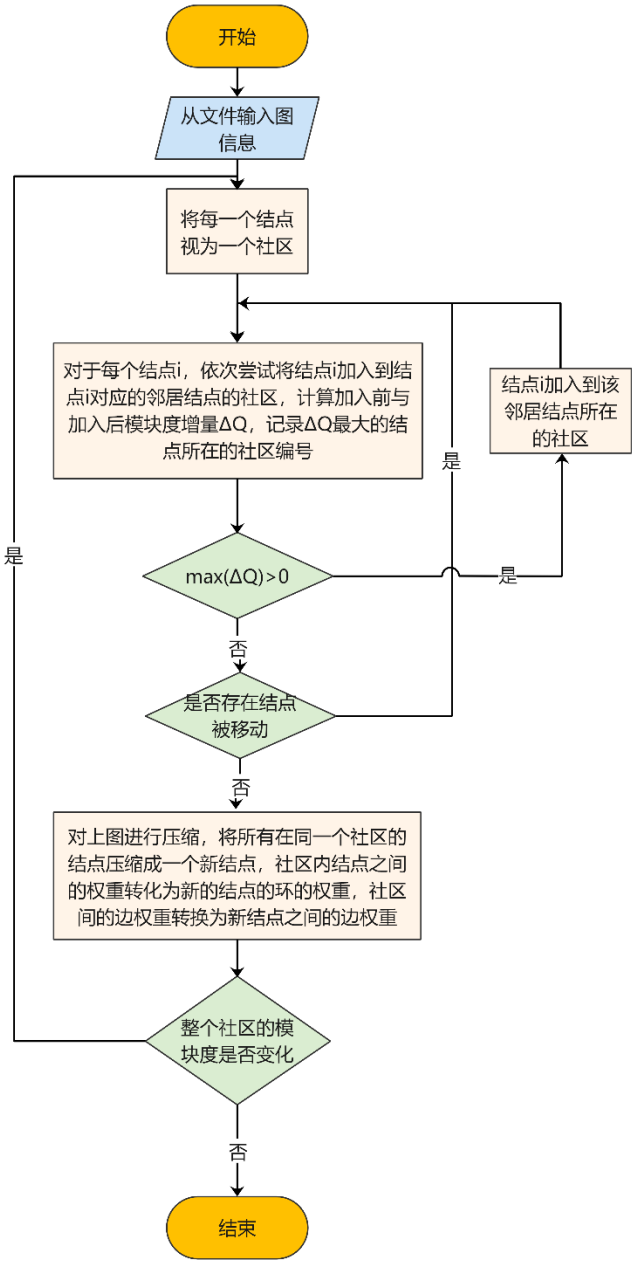

图 7 Louvain 算法流程图

2.3.2 伪代码

```
1.  def getBestPartition():
2.
3.      # 初始化：将输入的图转化为社区，返回社区以及边的权重（默认为 1.）
4.      community,ew= node2Community(graph)
5.      # 首先执行 Phase1
6.      community= Phase1(community,ew)
```

```
    best_modularity= caculateModularity(community,ew)

    partition=community
    new_community,new_ew=self.Phase2(community,ew)

    while True:
        new_community=Phase1(new_community,new_ew)
        modularity=caculateModularity(new_community,new_ew,param)

        if abs(best_modularity-modularity)<MIN_VALUE:
            # 如果 ΔQ 小于阈值，结束算法
            break

        best_modularity=modularity #得到当前划分结果的模块度
        partition=updatePartition(new_community,partition)
        #Phase2 步骤
        new_community,new_ew=Phase2(new_community,new_ew)

    return partition #得到社区划分结果
```

2.4 算法实现细节

2.4.1 语言

Python

2.4.2 环境

Pycharm+Python3.8
涉及到的包有：

表 2 Louvain 算法社区划分涉及到的包

包名	作用
networkx	读取并构建网络
matplotlib	绘图
permutation	计算
combination	计算
defaultdict	计算以及存储临时答案
time	记录运行时间
numpy	科学高效计算
communities	验证算法准确性（Louvain 算法邻接矩阵法的比较实验）

sklearn	Louvain 算法结果评估

2.4.3 关键步骤的实现细节

1. 数据结构：
   1) 网络信息 Gragh：

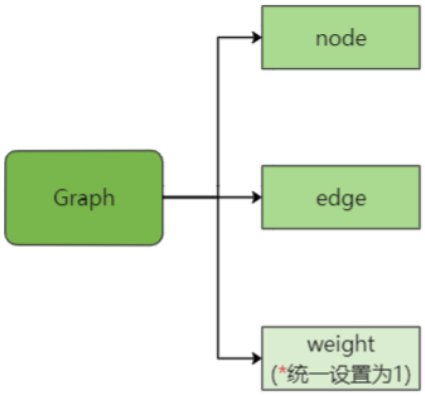

图 8 利用 networkx 存储网络结点以及边信息

   2) 社区信息 community:字典数据，用来存储当前划分的每个社区对应的结点
   3) 边信息 ew:存储结点连接以及对应边权信息（文中认为所有的边权均为 1.0）

2. 主要函数以及模块：
   - **数据加载：**
     loadGraph():从 txt 文件中加载结点连接数据，并返回 nx.Graph()格式的数据

   - **Louvain 算法实现：**

   1) Node2Community():将输入的图进行算法初始化，每一个结点为一个社区
   2) Phase1()：执行 Phase1
   3) Phase2()：执行 Phase2
   4) caculateModularity()：计算模块度得分
   5) updatePartition()：更新社区划分后的相关数据

   - **结果可视化**：
     将划分结果储存进字典，利用索引标签为不同社区的结点进行着色，利用 networkx 和 matplotlib.pyplot 联合对社区划分后的结果可视化

   - **结果评估：**
     Evaluate():利用 sklearn 自带的聚类算法评估函数对于社区划分结果进行评估

2.5 运行结果

2.5.1 小型网络

1. 数据集描述：

表 3 Dolphin 数据集特征描述

结点个数	边个数	自环个数	网络传递性
62	159	0	0.309

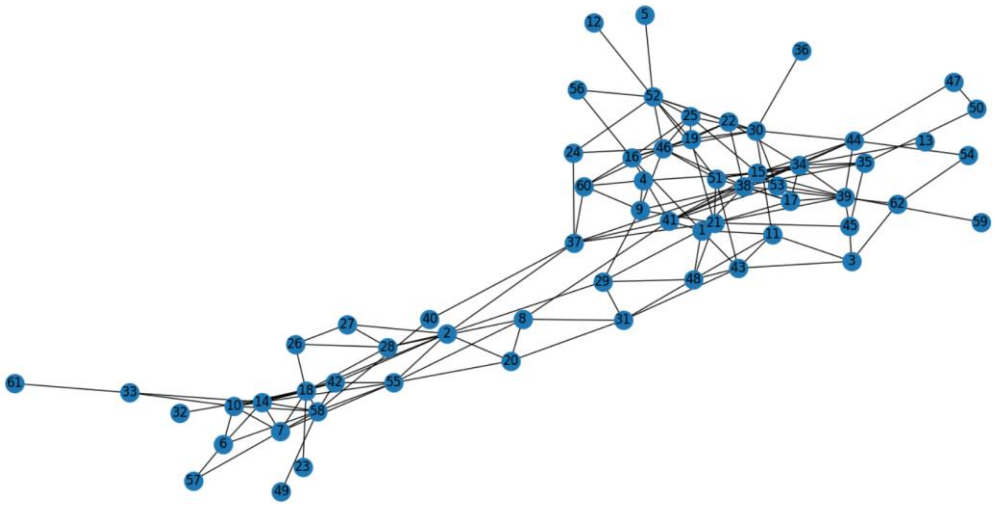

图 9 海豚交流社区原始无向图可视化

2. 结果

1) 基于链表的 LOUVAIN 算法：

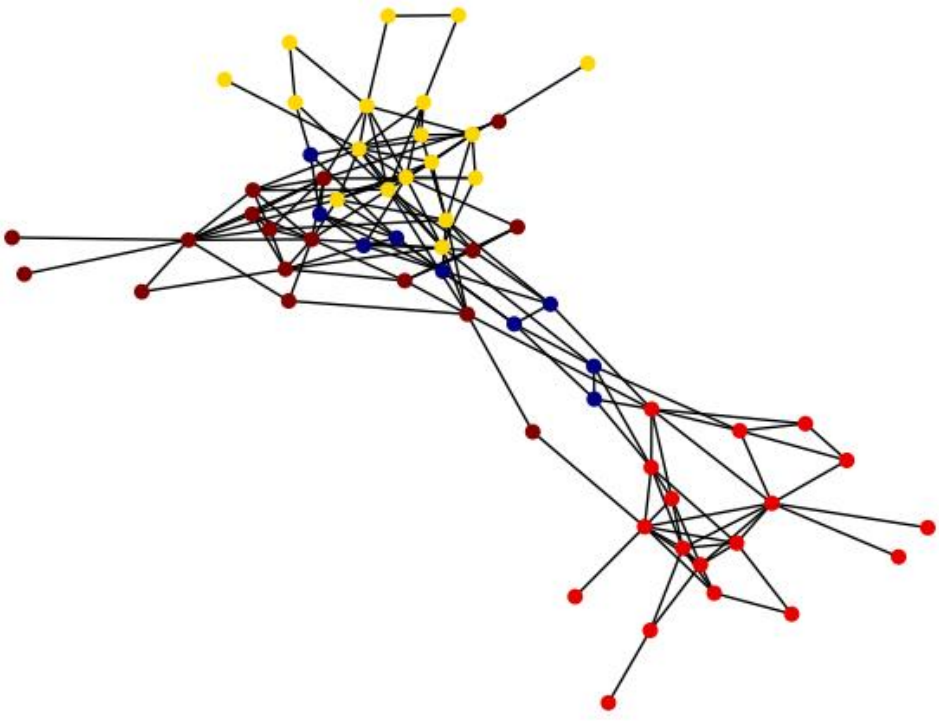

图 10 海豚交流社区原始 louvain 算法划分结果可视化

- **最终模型的划分结果**

  30 ['11', '1', '43', '48', '20', '29', '3', '8', '31']

  45 ['15', '41', '45', '62', '21', '38', '34', '13', '17', '35', '39', '44', '51', '53', '50', '59', '47', '54']

  52 ['16', '37', '9', '4', '60', '52', '5', '46', '30', '12', '25', '19', '56', '22', '24', '36', '40']

  50 ['18', '2', '27', '28', '42', '55', '10', '6', '14', '57', '58', '7', '33', '23', '26', '32', '61', '49']

  （注意* num 标记的为模型得出的社区编号）

  {0, 2, 7, 10, 42, 47, 19, 30}
  {32, 1, 5, 6, 9, 41, 13, 31, 48, 17, 22, 54, 56, 25, 26, 27, 60, 57}
  {3, 36, 39, 8, 59, 28}
  {35, 4, 11, 45, 15, 18, 51, 23, 21, 55, 24, 29}
  {33, 34, 37, 38, 40, 43, 12, 44, 14, 46, 16, 49, 50, 20, 52, 53, 58, 61}

- **模块度**

  Mod=0.519

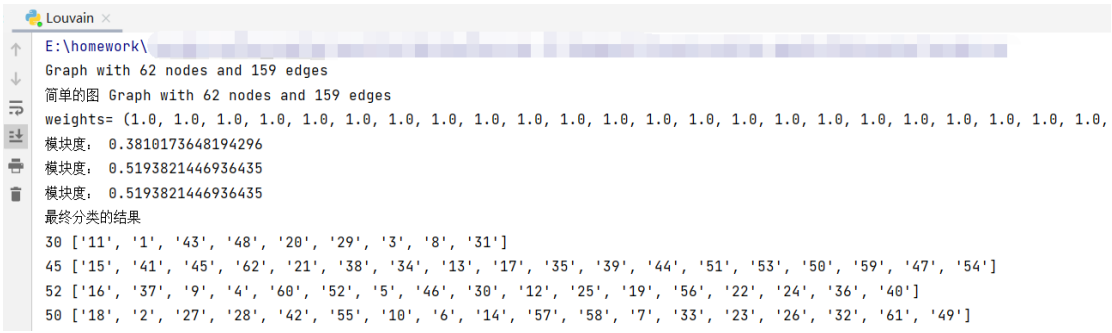

图 10 Louvain 算法对于 dolphin 数据集测试结果

3. 模型评估

最终划分出的社区的模块度为 0.519，在 0.3-0.7 之间，数值较高，可见 Louvain 算法的划分结果较为良好。

2.5.2 中型网络

1. 数据集描述

表 4 SNAP-Email-Eu-core 数据集特征描述

结点个数	边个数	自环个数	网络传递性
1005	16706	19	0.267

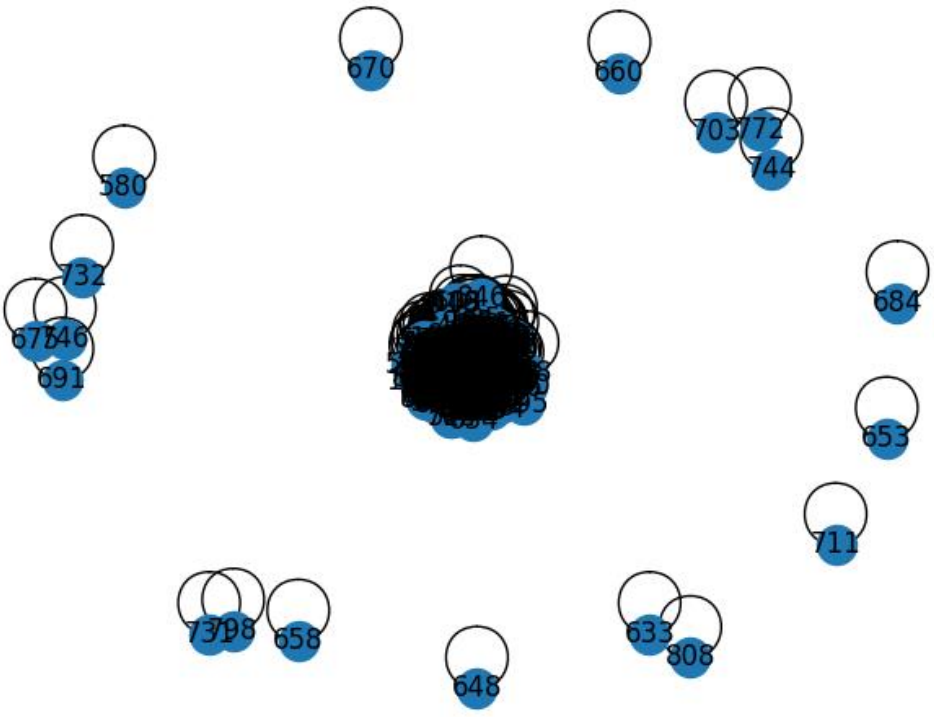

图 11 部门邮件交流原始无向图可视化

2. 结果

1) 基于链表的 LOUVAIN 算法：

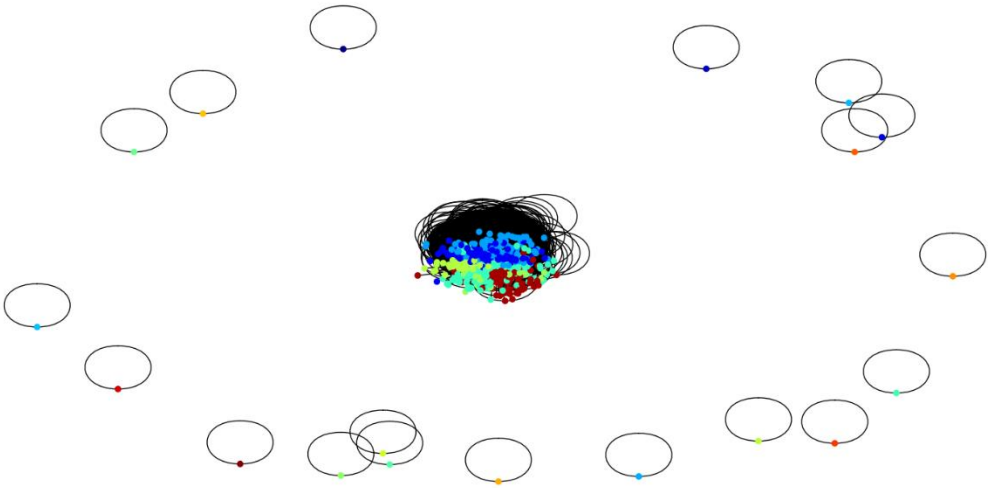

图 12 部门邮件交流原始 Louvain 划分结果可视化

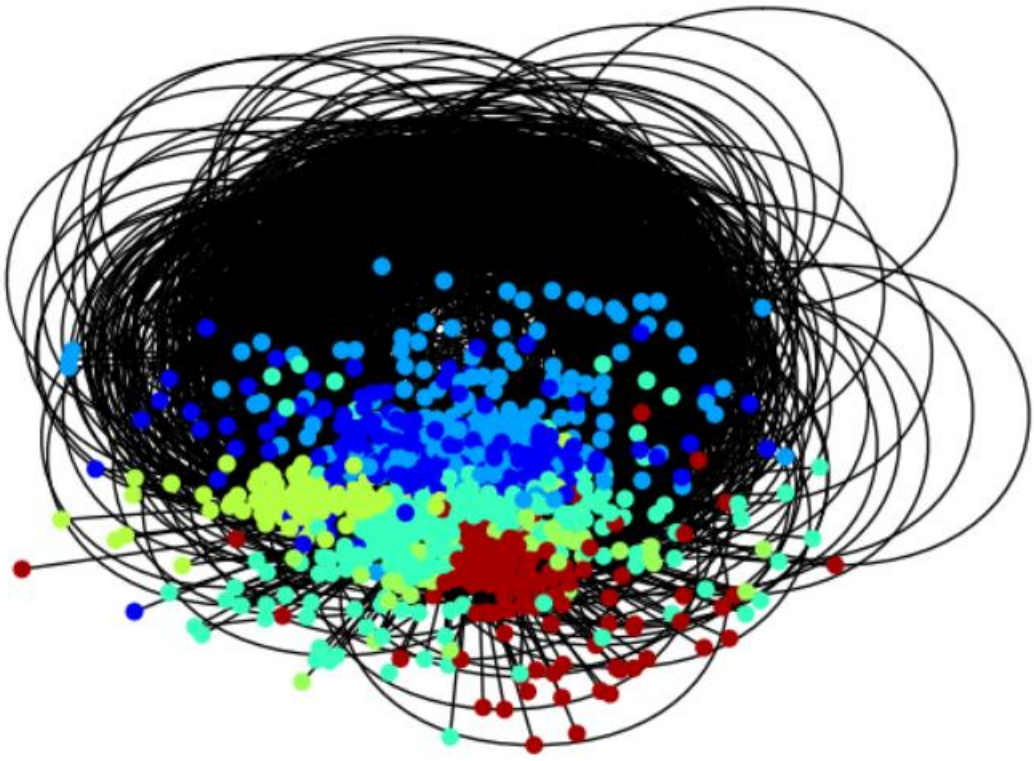

图 13 部门邮件交流原始 Louvain 划分结果（除去自环）可视化

- **最终模型的划分结果:**

330 [0, 1, 14, 17, 18, 52, 53, 60, 61, 65, 73, 74, 85, 103, 104, 120, 122, 130, 146, 148, 149, 150, 156, 157, 160, 176, 177, 178, 179, 180, 181, 182, 191, 214, 215, 218,

219, 220, 221, 224, 225, 226, 228, 231, 232, 248, 250, 257, 262, 268, 270, 275, 276, 277, 278, 283, 284, 289, 295, 296, 297, 307, 308, 309, 310, 312, 313, 314, 316, 317, 320, 321, 330, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 368, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 401, 402, 413, 414, 415, 425, 440, 449, 457, 468, 479, 483, 507, 511, 526, 537, 539, 556, 560, 562, 568, 574, 575, 595, 603, 605, 621, 628, 629, 632, 645, 650, 656, 659, 664, 668, 680, 681, 682, 692, 696, 697, 705, 719, 724, 726, 734, 741, 745, 749, 758, 761, 764, 775, 779, 781, 782, 791, 797, 810, 814, 824, 835, 839, 840, 841, 846, 851, 852, 857, 861, 869, 870, 871, 872, 873, 874, 903, 904, 905, 906, 907, 910, 916, 939, 943, 947, 948, 959, 960, 961, 979, 985, 998, 1002]

995 [2, 3, 4, 5, 6, 54, 55, 56, 57, 58, 59, 63, 88, 89, 102, 126, 131, 132, 137, 138, 158, 159, 174, 175, 192, 193, 194, 195, 208, 209, 210, 211, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 252, 271, 281, 285, 286, 302, 303, 304, 305, 319, 369, 373, 408, 411, 412, 481, 516, 517, 520, 528, 532, 552, 564, 571, 586, 587, 599, 601, 604, 610, 619, 622, 625, 630, 631, 634, 635, 636, 637, 639, 646, 665, 683, 685, 698, 712, 716, 717, 718, 737, 738, 743, 750, 755, 762, 763, 774, 784, 788, 803, 806, 807, 809, 812, 815, 826, 832, 842, 845, 849, 854, 859, 863, 864, 865, 866, 868, 876, 879, 880, 884, 886, 888, 898, 899, 901, 902, 921, 924, 926, 927, 928, 930, 931, 949, 963, 977, 982, 988, 990, 991, 993, 994, 995, 1001, 1004]

957 [7, 8, 9, 11, 12, 19, 43, 44, 141, 161, 213, 246, 247, 264, 265, 266, 267, 293, 324, 331, 332, 358, 359, 360, 362, 374, 406, 407, 421, 430, 441, 451, 452, 466, 487, 488, 496, 498, 499, 500, 501, 502, 503, 504, 505, 506, 510, 525, 529, 530, 533, 555, 558, 565, 566, 569, 570, 573, 602, 608, 616, 649, 661, 666, 672, 674, 699, 700, 707, 720, 729, 740, 754, 765, 778, 804, 805, 823, 827, 830, 833, 856, 893, 912, 913, 922, 950, 951, 956, 957, 967, 971, 972, 973, 975, 996]

643 [10, 16, 20, 21, 22, 42, 49, 50, 62, 66, 67, 68, 69, 70, 71, 72, 77, 78, 80, 81, 82, 83, 84, 87, 90, 91, 92, 105, 106, 107, 108, 109, 110, 111, 112, 117, 118, 121, 127, 142, 144, 145, 147, 152, 153, 154, 155, 162, 163, 166, 173, 183, 184, 186, 187, 188, 189, 190, 212, 217, 222, 223, 227, 230, 249, 253, 254, 255, 256, 258, 259, 260, 279, 282, 287, 288, 298, 299, 300, 306, 311, 315, 325, 326, 327, 328, 329, 355, 356, 357, 363, 364, 365, 366, 371, 372, 375, 400, 405, 410, 416, 418, 419, 420, 422, 424, 431, 432, 433, 434, 435, 453, 454, 456, 459, 460, 462, 463, 465, 467, 469, 471, 472, 473, 474, 475, 476, 477, 478, 480, 489, 490, 492, 495, 508, 509, 512, 513, 514, 518, 519, 524, 531, 536, 538, 540, 541, 546, 548, 549, 550, 551, 559, 561, 577, 578, 582, 589, 591, 594, 596, 597, 606, 607, 612, 613, 614, 615, 626, 627, 638, 641, 642, 643, 647, 652, 654, 663, 667, 669, 671, 673, 677, 678, 679, 690, 693, 695, 701, 702, 704, 710, 713, 715, 723, 727, 728, 730, 733, 736, 739, 742, 747, 748, 752, 756, 759, 767, 769, 771, 773, 780, 783, 786, 787, 792, 793, 796, 799, 802, 818, 821, 825, 828, 831, 834, 837, 850, 853, 855, 858, 877, 882, 885, 887, 889, 890, 891, 894, 896, 909, 911, 920, 923, 932, 934, 941, 942, 944, 945, 946, 952, 954, 958, 962, 966, 968, 969, 984, 989, 999, 1003]

813 [13, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 47, 48, 51, 64, 75, 76, 79, 86, 93, 94, 95, 96, 113, 114, 115, 116, 119, 123, 128, 129, 133, 134, 135, 136, 143, 151, 165, 167, 168, 169, 170, 171, 172, 196, 197, 198, 199,

200, 201, 202, 203, 204, 205, 206, 207, 229, 245, 251, 261, 263, 280, 290, 291, 292, 294, 318, 333, 336, 337, 338, 339, 340, 361, 367, 370, 399, 403, 409, 417, 423, 426, 427, 436, 437, 438, 439, 442, 443, 444, 445, 450, 455, 458, 464, 470, 482, 484, 485, 486, 491, 493, 494, 497, 515, 521, 522, 523, 527, 534, 535, 542, 543, 544, 545, 547, 553, 554, 557, 563, 567, 572, 576, 581, 583, 584, 585, 588, 590, 593, 598, 600, 609, 611, 620, 623, 655, 686, 688, 689, 694, 706, 714, 721, 722, 725, 751, 753, 757, 766, 776, 777, 785, 789, 790, 795, 801, 811, 813, 816, 817, 820, 822, 829, 836, 843, 844, 847, 848, 860, 862, 867, 875, 878, 881, 883, 892, 895, 897, 900, 908, 914, 915, 917, 918, 919, 925, 936, 938, 940, 953, 955, 964, 965, 970, 974, 976, 978, 980, 981, 983, 986, 987, 992, 997, 1000]

354 [15, 45, 46, 97, 98, 99, 100, 101, 124, 125, 139, 140, 164, 185, 216, 269, 272, 273, 274, 301, 322, 323, 334, 335, 353, 354, 404, 428, 429, 446, 447, 448, 461, 579, 592, 617, 618, 624, 640, 644, 651, 657, 662, 676, 687, 708, 709, 735, 760, 768, 770, 794, 800, 819, 838, 929, 933, 935, 937]

580 [580]

633 [633]

648 [648]

653 [653]

658 [658]

660 [660]

670 [670]

675 [675]

684 [684]

691 [691]

703 [703]

711 [711]

731 [731]

732 [732]

744 [744]

746 [746]

772 [772]

798 [798]

808 [808]

（注意* num 标记的为模型得出的社区编号）

- **模块度:**
  Mod=0.404

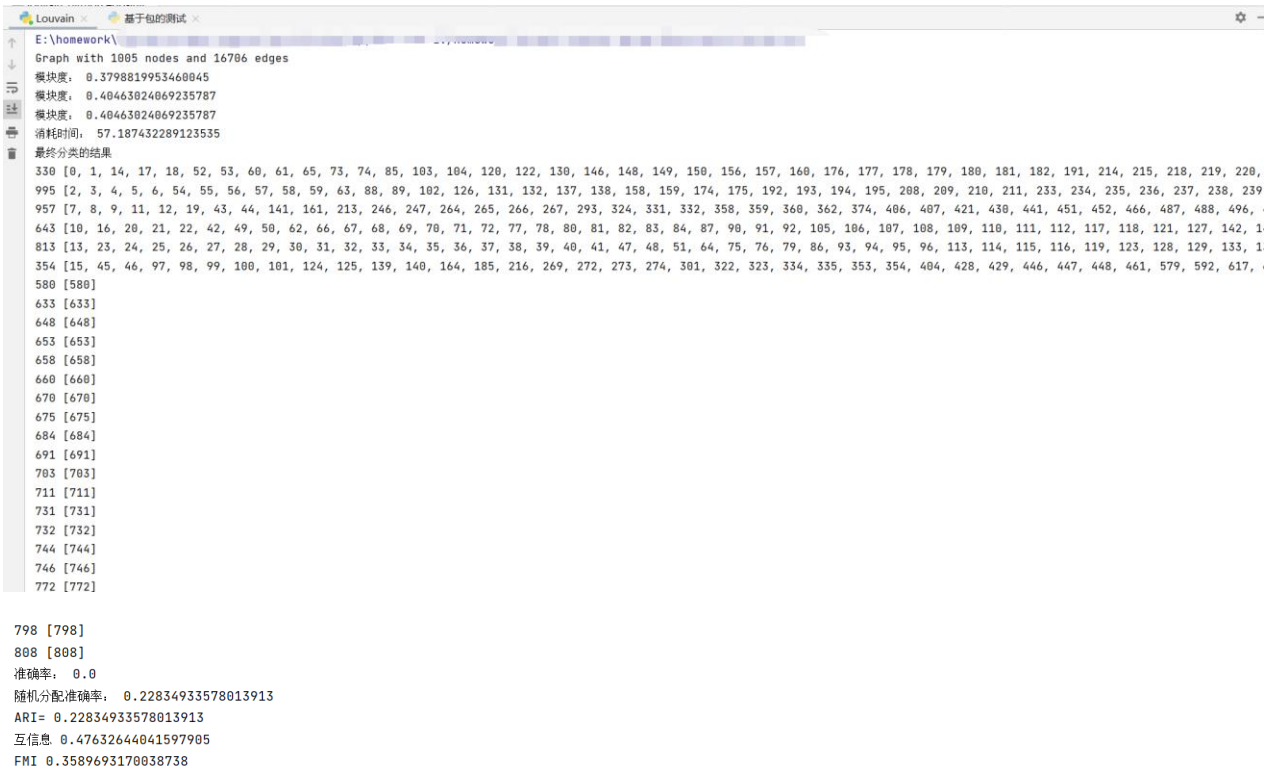

图 14 Louvain 算法测试 Emial 数据集结果

- **ACC:**

由于 Louvain 算法属于无监督的分类算法，其聚类得出的社区标签并没有具体含义（本文算法选择每一个社区中的某一个结点的编号作为该社区的编号），所以直接计算 acc 的答案为 0.

查阅了相关资料后，引入 ARI、FMI 以及互信息作为评估参数，其结果如下表

表 5 Emial 数据集社区划分结果检验

ARI	互信息	FMI	ACC
0.228	0.476	0.359	0.000

（注*无监督下标签的划分依据社区内部结点编号随机，导致直接用 label 数据测试 ACC 并没有实际意义）

3. 模型评估

Louvain 算法对于中型网络（email-Eu-core.txt）计算得出的结果模块度为 0.404，在 0.3-0.7 之间，属于可以接受的范围。较小型网络（small_dolphin.txt）模块度略有下降，由此可见数据集规模以及特点本身也会影响 Louvain 算法的划分效果。于此同时，在计算 ACC 的时候，由于无监督算法本身对于标签的划分的无意义性，使得直接计算 ACC 显然是有失偏颇的。在此我们引入 ARI、FMI 以及互信息作为评估参数，见表得出的结果尚可。

2.6 问题探讨

2.6.1 复杂性分析

Louvain 算法的实质是一种贪心算法，即将图的划分问题变相转化成求模块度最大化。Louvain 算法的主要时间复杂度消耗在 Phase1 和 Phase2 的迭代上面，假设结点数为 n 则时间复杂度为 O(nlogn).

由于使用的策略是贪心，求解结果上不一定是最优的，但是相较于其他图划分算法，其时间复杂度较小，适用于研究大型网络，因为它实现速度快，收敛快，高模块化输出。

2.6.2 正确率分析

本次实验中对于小型数据集 Dolphin 以及中型数据集 Email 都做了社区划分，值得注意的是，在有标签的 Email 数据集上，由于实验中对于社区划分的标签并不具有实际含义，划分出的社区数 (25) 也与实际社区数 (34) 有所出入。在进行正确率评估时，本文采用了 ARI、互信息 MI 以及 FMI 三种不同的聚类算法评估指标来对算法准确度进行更为客观的评价，发现各项指标分别为 0.228，0.476 与 0.359，可以证明算法在短时间内取得了较好的结果。在两个不同大小的数据集上，算法最终得到的模块度，分别为 0.519 与 0.404，均在 0.3~0.7 的理想范围内，因此 Louvain 实验结果较为良好。

3 实验总结与反思

3.1 实验总结

编程环境：由于科研训练，笔者目前正在学习一些 NLP 算法，所以经常要用到 python 的环境。本次实验也完全依赖 python3.8+pycharm 的环境平台，相较于 C++与 java 的同类算法，python 代码本身的时间效率确实略逊色于前者，但是由于其具有相较于前者更加丰富包与更为强大的可视化功能，使得在进行实验的时候可以更多地将时间倾注于算法本身而不用过多考虑如何将结果可视化的问题。

算法的时间效率直观求解：本次实验主要利用了 python 自带的 time 包来对于算法进行具体的时间维度的衡量与比较。对于数独问题的多解情况，在面对某些特殊的数独棋盘的时候，以普通 cpu 性能，完全跑出所有的解情况显然是不太现实的（如，完全空白的数独棋盘的解空间达到了 6.67×10 的 21 次方），于是本次实验采用建立子线程监控函数执行状态，限制函数运行时间并输出最终有限时间内跑出的结果总数，该方法的缺点是需要循环监控返回值，资源消耗较大，但是可以较为直观地展示出算法求解效率。

3.2 实验反思

数独问题：数独问题，本次实验中用到的是回溯法，可以有效解决解空间的重复遍历造

成的运算冗余问题。

Louvain 算法的社区划分：本次实验中利用了 networkx 包进行网络的数据封装，其目的主要是为了更好地对于结果进行可视化，然而这不可避免地导致了求解过程中网络地数据结构需要不断的变化，在一定程度上造成了计算资源的浪费。

4 参考文献

[1] Microsoft Word - H19 Rec backtrack examples.doc (stanford.edu)
[2]Vincent D Blondel,Jean-Loup Guillaume,Renaud Lambiotte,Etienne Lefebvre. Fast unfolding of communities in large networks[J]. Journal of Statistical Mechanics: Theory and Experiment,2008,2008(10):
[3]http://www.cse.cuhk.edu.hk/~cslui/CMSC5734/newman_community_struct_networks_phys_rev.pdf
[4] CS224W-图神经网络 笔记 4.2：Community Structure in Networks - 网络中社区的挖掘算法——Louvain 算法 - HD 的博客 | HD Blog (whdi.top)
[5] python 复杂网络结构可视化——matplotlib+networkx - 知乎 (zhihu.com)

5 附录

5.1 代码文件说明

```
+--代码
| +--data
| | +--email-Eu-core.txt                          邮件往来数据
| | +--email-Eu-core-department-labels.txt        邮件标签数据
| | +--dolphin.gml                                NetWorkx 生成的海豚网络
| | +--email.gml                                  NetWorkx 生成的邮件网络
| | +--small_dolphin.txt                          海豚交流数据
| +--result
| | +--Dol_original.png                           海豚网络初始可视化
| | +--Dol_res.png                                海豚网络社区划分结果
| | +--Email_original.png                         邮件网络初始可视化
| | +--Email_res.png                              邮件网络社区划分可视化
| +--code
| | +--作业 1_数独
| | | +--简单解法_ghy.py                          数独简单遍历实验源码
| | | +--回溯法_ghy.py                            数独回溯法源码
| | +--作业 2_Louvain
| | | +--Louvain.py                               Louvain 算法源码
```

| | | +--NetWorkx 获取网络结构信息.py 数据集信息获取