学生学号	0122010870125	实验课成绩	

武漢理工大學

学 生 实 验 报 告 书

实验课程名称	编译原理
开 课 学 院	计算机与人工智能学院
指导教师姓名	王云华
学 生 姓 名	雷裕庭
学生专业班级	软件 sy2001 班

2022 -- 2023 学年 第 一 学期

实验课程名称：编译原理

实验项目名称	DFA 的化简			实验成绩	
实验者	雷裕庭	专业班级	软件 sy2001	组别	词法解譯のフォーリズム
同组者	雷裕庭 熊壮 张艺晨 程英豪			实验日期	2022 年 11 月 17 日

第一部分：实验分析与设计（可加页）

一、 实验内容描述（问题域描述）

【问题描述】DFA 化简问题的一种描述是：

编写一个程序，输入一个确定的有穷自动机（DFA），输出与 DFA 等价的最简的确定有穷自动机（DFA）。

【基本要求】

设置 DFA 初始状态 X，终态 Y，过程态用数字表示：0 1 2 3…

【测试用例】

测试数据： X X-a->0 X-b->1
Y Y-a->0 Y-b->1
0 0-a->0 0-b->2
1 1-a->0 1-b->1
2 2-a->0 2-b->Y

输出结果应为：X X-a->0 X-b->X
Y Y-a->0 Y-b->X
0 0-a->0 0-b->2
2 2-a->0 2-b->Y

二、实验基本原理与设计（包括实验方案设计，实验手段的确定，试验步骤等，用硬件逻辑或者算法描述）

【实验原理】

任何正规语言都有一个唯一的状态数目最少的 DFA

DFA M 的化简是指：寻找一个状态数比 M 少的 DFA M’，使得 L(M)=L(M’)

有穷自动机的多余状态：从自动机的开始状态出发，任何可识别的输入串也不能到达的状态

化简了的 DFA M’ 满足两个条件：

1.没有多余状态 2.没有两个状态是等价的

【求解步骤】

① 将 DFA M 的状态集 Q 分划成两个子集：终态集和非终态集；

②如果面对某个输入符号得到的后继状态不属于同一个子集，则将 G 进一步划分；

③ 重复②直到不再产生新划分；

④ 在每个子集中选一个状态作代表，消去其他状态，得到最少状态的等价 DFA M’。

三、主要仪器设备及耗材

PC 机

PyCharm 编译器 Python 包:copy queue combinations

第二部分：实验调试与结果分析（可加页）

一、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

下面是主要的程序设计说明

1.流程图

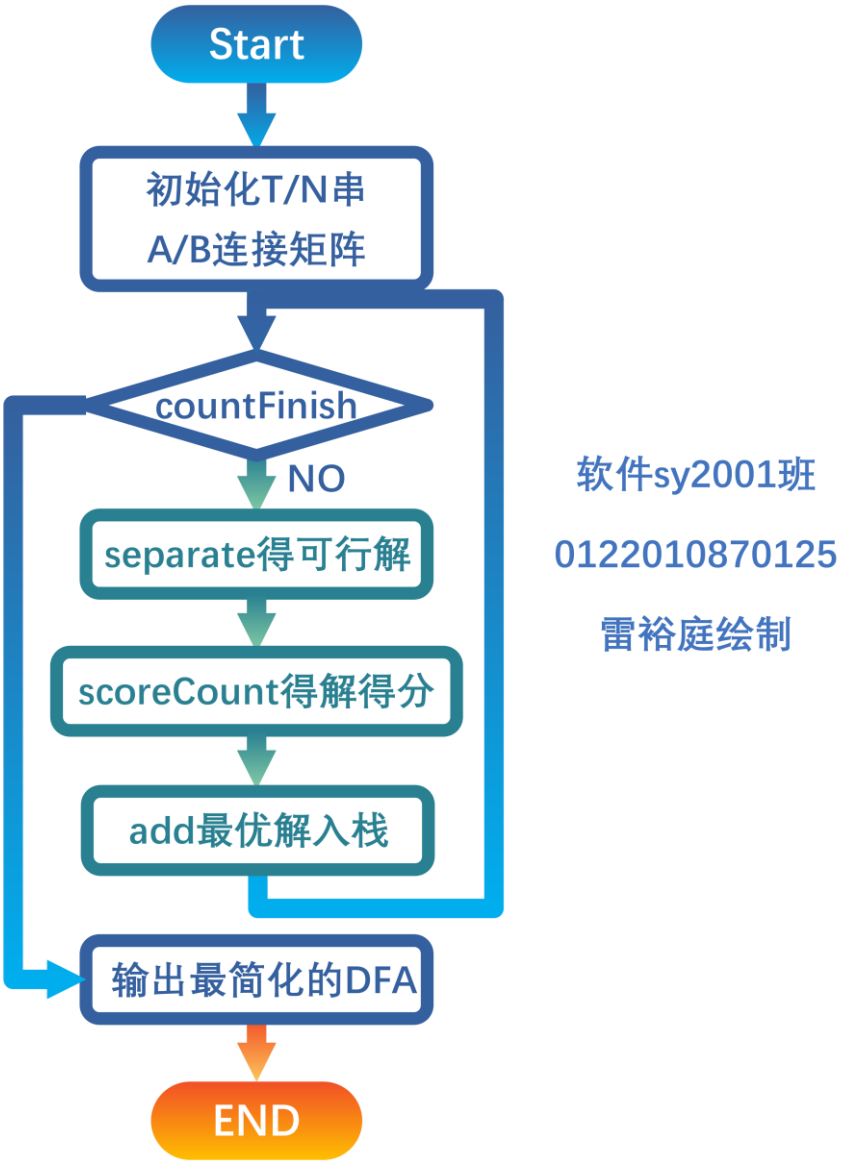
本程序的主要流程如下，

1.首先初始化主要数据结构和连接矩阵 MatrixA MatrixB

2.在未达到全部串都 均等价 的情况下执行第三部

3.通过基于 A*算法的 separate 函数不断的划分未均等价的串

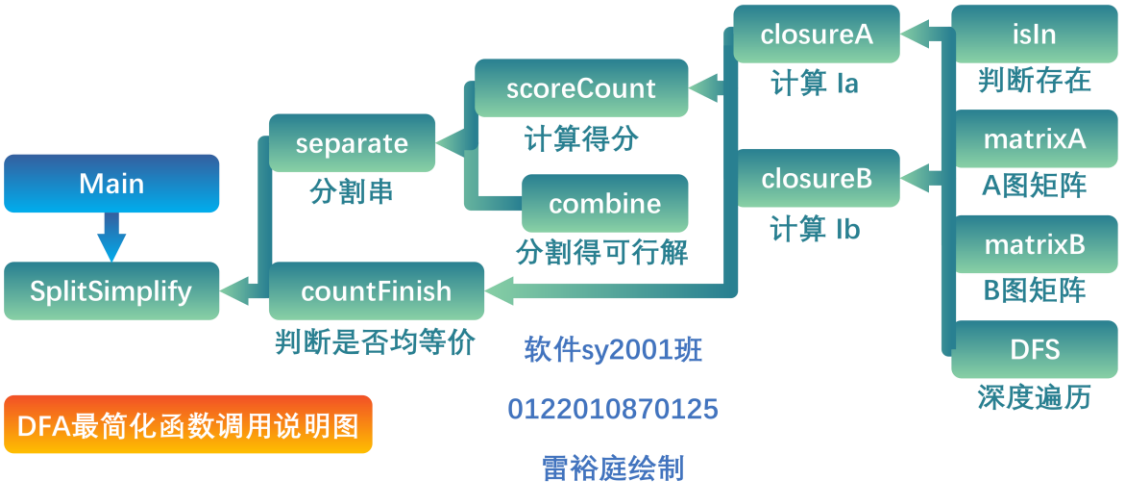
4.在全部串都 均等价 的情况下输出结果 结束程序



2.函数结构

基于以上要求的函数划分如下

具体的实现已经在图片中进行说明



3.1 A*算法说明

在搜索图中有很多叶节点，究竟应该对哪个节点进行扩展，一个直观的方案是，如果某个叶节点 n 距离初始节点 S 的距离再加上节点 n 到目标节点 G 的最小距离之和最小，那么该节点处在最短路径上，应该优先扩展。我们用 g(n)表示 S 到 n 的最短路径距离，用 h(n)表示 n 到 G 的最短路径距离，则从 S 经过 n 到达 G 的总距离 f(n)为：

$$f(n) = g(n) + h(n)$$

如果我们选择 f(n)最小的叶节点进行扩展，将保证搜索效率最高。

但是在搜索过程中，我们还没有找到一条从 n 到 G 的最短路径，因此也就不知道 h(n)是多少。

为解决该问题，一种方案是用当前得到的 S 到 n 的距离 g’(n)代替未知的最短距离 g(n)，并用一个估计值 h’(n)代替 n 到 G 的最短距离 h(n)，基于这两个近似值得到 f(n)的估计

$$f’(n)=g’(n)+h’(n)$$

，并基于 f’(n)进行搜索。

算法首先从初始节点 S 开始，每次选择一个 f’(n)值最小的叶节点进行扩展，直到扩展出目标 G 且 f’(G)在所有叶节点中取值最小为止。在搜索过程中，如果遇到多条路径到达同一个节点的情况，需要更新从 S 到达该节点的最短路径估计 g’(n)。

上述算法被称为 A 算法。在 A 算法中，对 h’(n)没有明确限制，只要符合直觉即可，因此 h’(n)可能比 h(n)小，也可能比 h(n)大，其中 h(n)为 n 到 G 的最短路径。A 算法不能保证找到的路径是最短路径。然而，如果对 h’(n)加以限制，使得对于任何一个叶节点 n，总有

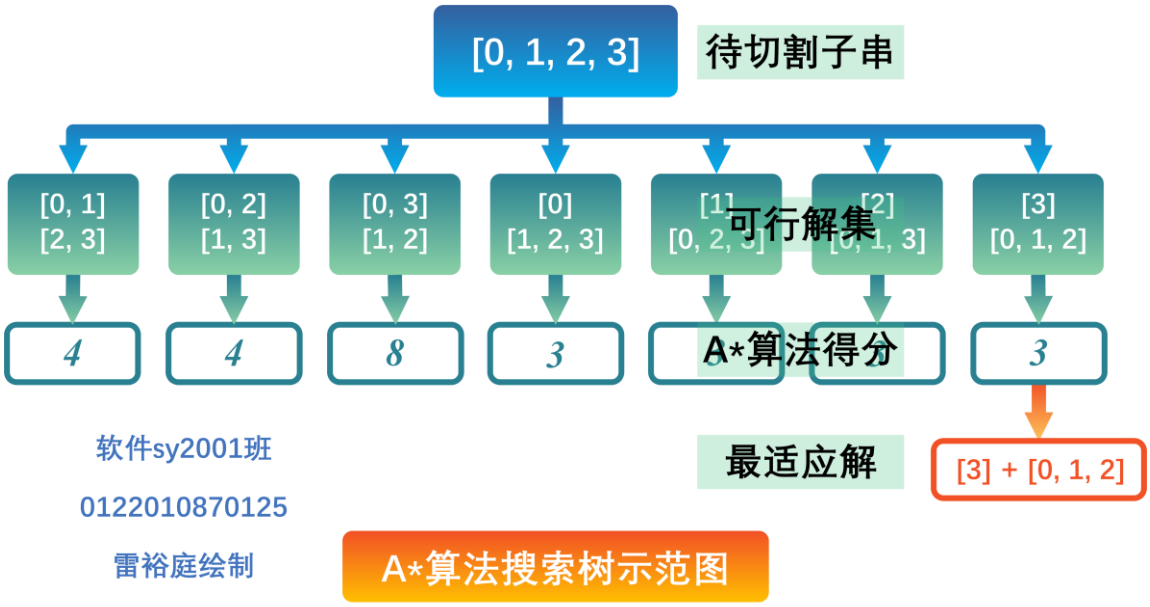
$$h’(n)<=h(n)$$

，那么该算法找到的路径一定是最短的，此时 A 算法被称为 A*算法。

3.2 A*算法与本实验的结合

在本实验中，组串的切割形式在一开始是难以被判断的，且在数据量较大的情况下会变为 np 难问题。因此本实验的代码编写中使用 **A*算法**来实现寻找最佳切割方法的目的。

A*算法对于实验用例中的**[0,1,2,3]**串的切割效果和计算得分如下



3.5 A*算法分数计算函数形式

A*算法的一大难点是难以寻找出较好的分数计算函数来选择最优解。单纯的只考虑单维度的计算函数会导致每次的搜索会偏向于局部最优解，而忽视了之后多步的可能情况。

为此本算法设计评价分数如下，同时基于当前未完成均等价串的数量和剩余串的长度两大因素。经过实验证明，该函数可以较好的适应实验的案例。

注:本函数计算的指标为极小化指标，即分数约低，代表情况效果越好

$$Score+=len(List[j])+min(len(List[i]["first"]),len(List[i]["second"]))$$

极小化指标+=均等价串长度+Min(切割串中各串长度)

4. 主要实验代码

```
'''
任务日志
软件 sy2001 班雷裕庭编写
11.11 开始编写
11.11 晚完成了除分割方法以外的内容  局部功能测试完成
11.12 无思路  暂停编写
11.13 使用 A 算法实现功能
11.13 晚  完成编写  功能测试通过
'''
import copy
import queue
from itertools import combinations
```

```
'''
下面是复用自实验二的部分函数 功能和实现方法不再做赘述
'''
# 从 dict 文件中获得 A 相关的连通图
def matrixA(input):
    maxNum=0
    for i in range(len(input)):
        maxNum=max(max(maxNum,input[i]['start']),max(maxNum,input[i]['next']))
    maxNum+=1
    matrix = [[0] * maxNum for _ in range(maxNum)]
    for i in range(len(input)):
        if input[i]['c']=='a':
            matrix[input[i]['start']][input[i]['next']] = 1
    return matrix
# 从 dict 文件中获得 B 相关的连通图
def matrixB(input):
    maxNum=0
    for i in range(len(input)):
        maxNum=max(max(maxNum,input[i]['start']),max(maxNum,input[i]['next']))
    maxNum+=1
    matrix = [[0] * maxNum for _ in range(maxNum)]
    for i in range(len(input)):
        if input[i]['c']=='b' :
            matrix[input[i]['start']][input[i]['next']] = 1
    return matrix
#实现 DFS
def DFS(matrix,n):
    out=set()
    stack=queue.Queue(maxsize=100)
    lenth=len(matrix)
    out.add(n)
    stack.put(n)
    #初始化栈
    while stack.qsize()>0:
        m=stack.get()
        for j in range(lenth):
            if matrix[m][j] == 1:
                if out.__contains__(j)!=True:
                    out.add(j)
                    stack.put(j)
    return    out
'''
因为不再存在空字符 对 closure 的计算自然有略微的修改
'''
```

```
#closureA:计算 set 的 Ia closure 集合 并输出为 set
def closureA(input:set,matrixA:list):
    lenth= len(matrixA)
    temp=copy.deepcopy(input)          #原始集合   为避免浅拷贝数据污染
    out=set()
    while temp:
        tempInt = temp.pop()
        for i in range(lenth):
            if matrixA[tempInt][i]==1:
                out.add(i)
    return out
#closureB:计算 set 的 Ib closure 集合 并输出为 set
def closureB(input:set,matrixB:list,):
    lenth = len(matrixB)
    temp = copy.deepcopy(input)  # 原始集合   为避免浅拷贝数据污染
    out = set()
    while temp:
        tempInt = temp.pop()
        for i in range(lenth):
            if matrixB[tempInt][i] == 1:
                out.add(i)
    return out
def getMax(input):
    maxNum=0
    for i in range(len(input)):
        maxNum=max(max(maxNum,input['start']),max(maxNum,input['next']))
    return maxNum
'''
若前者在后者 给予 True
'''
def isIn(inputSet,setList):
    lenth=len(setList)
    for i in range(lenth):
        if set(inputSet).issubset(set(setList[i]["element"])):
            return True
    return False

'''
combine:函数用于(在指定一边元素数量的情况下)生成全部的二分组合的可能性
输出:       数据结构表现为:      ({"first":[],"second":[]}*n)
注:         在 n=lenth/2 的情况下       会出现一半的组重复
'''
def combine(tempList: list(),groupSize):
    out=list()
```

```
    firstList=list(combinations(tempList, groupSize))
    for i in range(len(firstList)):
        tempSecond=set(copy.deepcopy(tempList))
        for j in range(groupSize):
            tempSecond.remove(firstList[i][j])
        out.append({"first":list(firstList[i]),"second":list(tempSecond)})
    return out

'''
scoreCount:基于 combine 的分割数组和已有的数组 计算 closure 集合是否满足情况
输入:                A 连接矩阵 B 连接矩阵 已有数组 尝试集
scoreList:          中途储存的评分集，评分越低越好(分数为还需要修改的组的长度的相加)
'''
def scoreCount(mA,mB,setList,tryList):
    scoreList=list()
    lenth=len(tryList)
    for i in range(lenth):
        #创建全集
        tempSetList=copy.deepcopy(setList)
        tempSetList.append({'class': 'T','element': tryList[i]["first"]})
        tempSetList.append({'class': 'T','element': tryList[i]["second"]})
        #计算
        tempScore=0
        for j in range(len(tempSetList)):
            temp1 = closureA(tempSetList[j]["element"], mA)
            temp2 = closureB(tempSetList[j]["element"], mB)
            if isIn(temp1, tempSetList) == False :

tempScore+=(len(tempSetList[j])+min(len(tryList[i]["first"]),len(tryList[i]["second"])))
            if isIn(temp2, tempSetList) == False :
                tempScore
+=( len(tempSetList[j])+min(len(tryList[i]["first"]),len(tryList[i]["second"])))
        scoreList.append(tempScore)
    return scoreList

'''
通过一个 A*算法实现          通过分割集合尽可能的增加 finishedNum 集合
解树:广度遍历获得   例:对于 4 元组  4*3 (层数:n//2)
A*的激励函数          best:两子集均可被 finish
                    middle:一个可 finish  一个不可 finish
                    worst:均不可 finish
输出:dict 数据结构的子集  out1 out2            分割之后全图的情况由更上一级进行计算
其他要求:过程隔离，避免数据污染，不修改输入数据
'''
```

```
def separate(mA,mB,setList,tempList):                    #输入 matrixA matrixB 当前集合情
况
    print("当前切割的组为")
    print(tempList["element"])
    orignalList=copy.deepcopy(tempList["element"])
    layerNum=len(orignalList)//2
    tryList=list()

    #计算全部的可行性
    for i in range(layerNum):
        tryList+=combine(orignalList, i+1)
    print("当前的二分组为")
    for i in range(len(tryList)):
        print(tryList[i])
    # 计算
    scoreList=scoreCount(mA, mB, setList, tryList)
    print("当前的二分组得分为")
    print(scoreList)
    #择优选择输出
    minNun=0
    minValue=scoreList[0]
    for i in range(len(scoreList)):
        if scoreList[i]<=minValue:
            minValue=scoreList[i]
            minNun=i

    return tryList[minNun]['first'],tryList[minNun]['second']

def test(input):
    mA = matrixA(input)
    mB = matrixB(input)
    setList=list()
    temp1 = {"class": 'T', "element": [4]}
    setList.append(temp1)
    tempList={"class": 'T', "element": [0,1,2,3]}
    separate(mA,mB,setList,tempList)

def countFinish(mA,mB,setList):
    finishedNum=len(setList)
    for i in range(len(setList)):
        temp=setList[i]["element"]
        temp1 = closureA(temp, mA)
        temp2 = closureB(temp, mB)
        if isIn(temp1, setList) == False or isIn(temp2, setList) == False:
```

```
                finishedNum-=1
    return finishedNum

'''
SplitSimplify 主逻辑计算函数
算法设计如下:
        1.首先区分出终结符集 T 和 非终结符集 U 两个集合均通过 dict 的方法储存在一个 list 中
                数据结构 [ {"class":'T',"cuttable":True,"element":[1,2,3]}*N   etc ]
        2.接下来就不断的访问 cuttable 的 dict 的 element，通过已有的 closureB 函数判断其是否要继续切割
'''

def SplitSimplify(input):
    # 初始化矩阵
    mA = matrixA(input)
    mB = matrixB(input)
    print("matrixA" + str(mA))
    print("matrixB" + str(mB))
    setNum=2                                    #集合数量
    finishedNum=0                               #不可分集合数量
    setList=list()                              #用于储存全部状态的集合  允许对被分割的集合实施删除操作

    #预处理 区分为 T 和 N
    temp1 = {"class": 'T', "element": [4]}
    temp2 = {"class": 'N', "element": [0,1,2,3]}
    setNum = 2
    finishedNum=1

    setList.append(temp1)
    setList.append(temp2)

    #开始计算
    while setNum>finishedNum:
        setNum = len(setList)
        finishedNum = countFinish(mA, mB, setList)
        #勉为其难的用 list 实现了一下一下 queue 的操作 实在不喜欢使用太多引申类型
        temp=copy.deepcopy(setList[0])
        del setList[0]
        print("出栈"+str(temp["element"]))
        #可分割
        if len(temp['element'])>1:
            temp1 = closureA(temp["element"],mA)
```

```
                temp2 = closureB(temp["element"],mB)
                #需分割
                if isIn(temp1,setList)==False or isIn(temp2,setList)==False:
                    #调用 A*算法开始分割
                    print("调用 A*算法开始分割")
                    out1,out2=separate(mA, mB, setList, temp)
                    #加入新的元素
                    setList.append({"class": 'T',    "element": out1})
                    setList.append({"class": 'T',    "element": out2})
                    setNum=len(setList)
                    finishedNum=countFinish(mA,mB,setList)
                    print(setNum,finishedNum)
                else :
                    setList.append(copy.deepcopy(temp))
            else:
                setList.append(copy.deepcopy(temp))
    return setList

if __name__ == '__main__':

    #注:为方便计算 已经将 XY 转换为数字 全部的原数字位做+1 处理
    input1 = [
        {'start': 0, 'next': 1, 'c': 'a'},
        {'start': 0, 'next': 2, 'c': 'b'},
        {'start': 4, 'next': 1, 'c': 'a'},
        {'start': 4, 'next': 2, 'c': 'b'},
        {'start': 1, 'next': 1, 'c': 'a'},
        {'start': 1, 'next': 3, 'c': 'b'},
        {'start': 2, 'next': 1, 'c': 'a'},
        {'start': 2, 'next': 2, 'c': 'b'},
        {'start': 3, 'next': 1, 'c': 'a'},
        {'start': 3, 'next': 4, 'c': 'b'}
    ]
    input2=[
        {'I': {0, 1, 4}, 'Ia': {1, 2, 4}, 'Ib': {1, 4}},
        {'I': {1, 4}, 'Ia': {1, 2, 4}, 'Ib': {1, 4}},
        {'I': {1, 2, 4}, 'Ia': {1, 2, 4}, 'Ib': {1, 3, 4}},
        {'I': {1, 3, 4}, 'Ia': {1, 2, 4}, 'Ib': {1, 4, 5}},
        {'I': {1, 4, 5}, 'Ia': {1, 2, 4}, 'Ib': {1, 4}},
    ]
    out=SplitSimplify(input1)
    print(out)
```

二、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）

1.可行解的获得 和 分数的计算

```
出栈[0, 1, 2, 3]
调用A*算法开始分割
当前切割的组为
[0, 1, 2, 3]
当前的二分组为
{'first': [0], 'second': [1, 2, 3]}
{'first': [1], 'second': [0, 2, 3]}
{'first': [2], 'second': [0, 1, 3]}
{'first': [3], 'second': [0, 1, 2]}
{'first': [0, 1], 'second': [2, 3]}
{'first': [0, 2], 'second': [1, 3]}
{'first': [0, 3], 'second': [1, 2]}
{'first': [1, 2], 'second': [0, 3]}
{'first': [1, 3], 'second': [0, 2]}
{'first': [2, 3], 'second': [0, 1]}
当前的二分组得分为
[3, 3, 3, 3, 4, 4, 8, 8, 4, 4]
```

```
出栈[0, 1, 2]
调用A*算法开始分割
当前切割的组为
[0, 1, 2]
当前的二分组为
{'first': [0], 'second': [1, 2]}
{'first': [1], 'second': [0, 2]}
{'first': [2], 'second': [0, 1]}
当前的二分组得分为
[3, 0, 3]
```

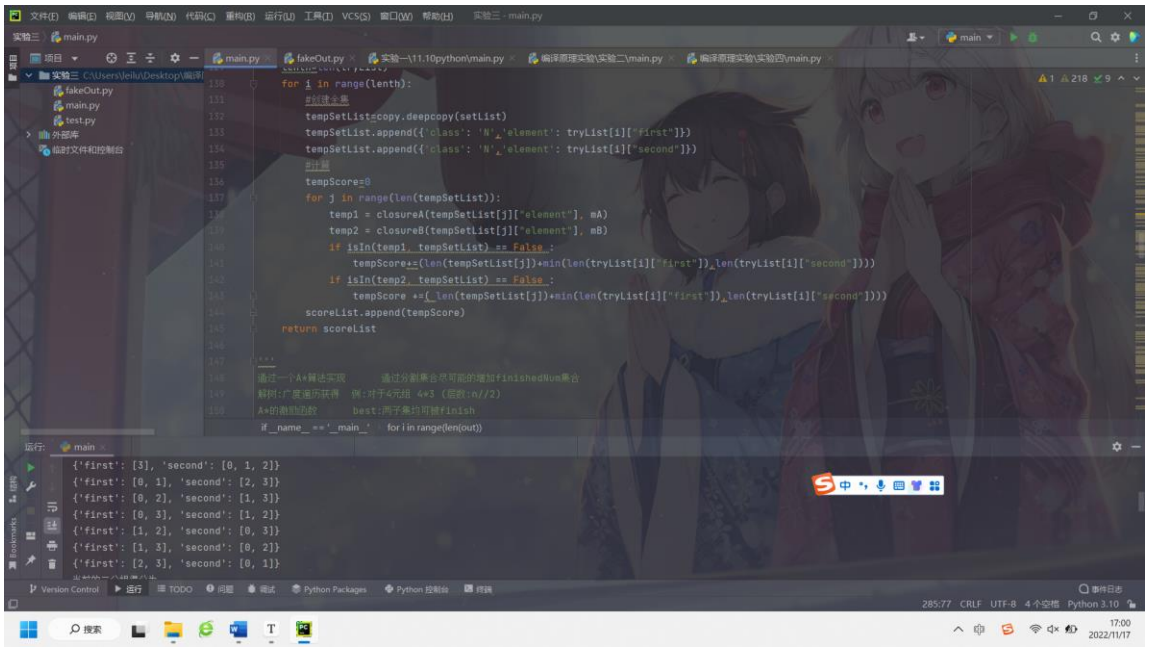
2.最后的分组串效果

```
函数计算结束，下面开始分组输出结果
串组:    [4]类型:    T
串组:    [3]类型:    N
串组:    [1]类型:    N
串组:    [0, 2]类型:    N
```

3.整理后输出结果

```
计算结果为
X X-a->0 X-b->X
  Y Y-a->0 Y-b->X
0 0-a->0 0-b->2
2 2-a->0 2-b->Y
```

4.部分函数展示



三、实验小结、建议及体会

【实验小结】

通过了这次的实验，本人学会了解了 DFA 的化简操作，并且通过了实践将自己的所学进行的实践。

同时也进一步的熟练了 Python 的使用。

【建议】

1.第三个实验的算法相对于其他几个实验过于随机化了。

并没有一个统一的算法，使得代码的书写过程过于随机。

2.希望可以在实验的前期得到对于不同实验难度的大致说明

【体会】

写算法还是有点有趣的，不过部分同学应该不太喜欢