中山大学计算机学院

人工智能

本科生实验报告

(2022 学年春季学期)

课程名称：Artificial Intelligence

教学班级	计科 1 班	专业（方向）	计算机科学与技术
学号	21307077	姓名	凌国明

一、实验题目

一阶谓词逻辑归结算法

编写程序，实现一阶逻辑归结算法，并用于求解给出的三个逻辑推理问题，要求输出按照如下格式：

1. $(P(x), Q(g(x)))$
2. $(R(a), Q(z), \neg P(a))$
3. $R[1a, 2c]\{X = a\}(Q(g(a)), R(a), Q(z))$

“R” 表示归结步骤
“1a” 表示第一个子句(1-th)中的第一个 (a-th)个原子公式，即 P(x)
“2c” 表示第二个子句(1-th)中的第三个 (c-th)个原子公式，即 ¬P(a)
“1a” 和 “2c” 是冲突的，所以应用最小合一 {X = a}

任务一、 Aipine Club

A(tony)
A(mike)
A(john)
L(tony, rain)

L(tony, snow)
(¬A(x), S(x), C(x))
(¬C(y), ¬L(y, rain))
(L(z, snow), ¬S(z))
(¬L(tony, u), ¬L(mike, u))
(L(tony, v), L(mike, v))
(¬A(w), ¬C(w), S(w)

任务二、 Graduate Student

GradStudent(sue)
(¬GradStudent(x), Student(x))
(¬Student(x), HardWorker(x))
¬HardWorker(sue)

任务三、 Block World

On(aa,bb)
On(bb,cc)
Green(aa)
¬Green(cc)
(¬On(x,y), ¬Green(x), Green(y))

二、实验内容

1、算法原理

子句集表示方法

子句集是一种便于计算机处理的表达形式。这种形式下，每一条子句对应着一个列表，列表中的每一个元素是一个项（离散数学中项的定义），同一列表的不同元素代表同一子句的不同谓词，**同一子句的不同谓词之间是析取关系**；不同列表代表不同的子句，**不同的子句是合取关系。**

归结算法

1. 将输入的一组公式**化为子句集合**，每个子句是由一个或多个谓词和它们的参数组成的。
2. 选择两个子句，使用**合一算法**找到两个子句中的某些谓词和参数的匹配项，并将它们替换为新的变量

3. 如果两个子句的某些谓词**在合一意义下是互补的**（即正负性相反，其他一致），则消去互补的谓词，然后两个子句**合成为一个新子句**
4. 将新的子句加入子句集合中，并重复步骤234，**直到找到一个空子句**，表示原始公式不可满足；**或不再产生新子句**，表示原始公式可以满足。

归结原则

对于子句集：

$$((P1, C_1), (\neg P, C_2), \dots)$$

归结原则：

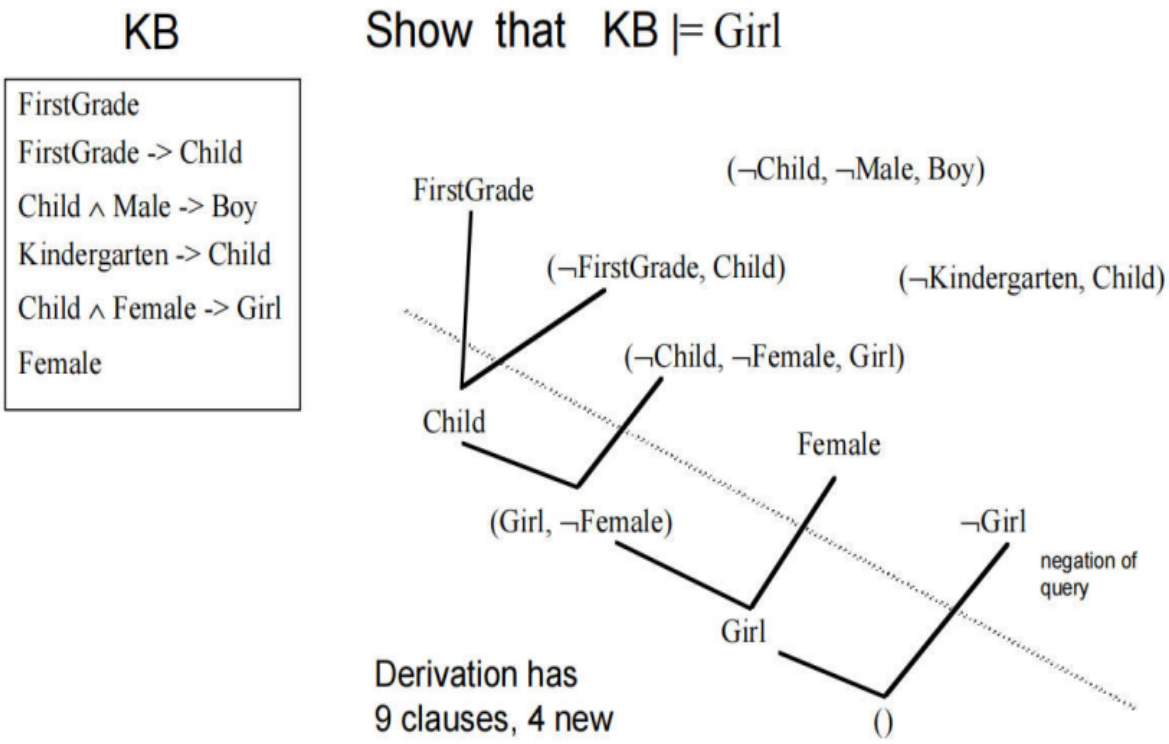
$$(P1, C_1) \wedge (\neg P, C_2) \Rightarrow (C_1, C_2)$$

它的特殊形式：

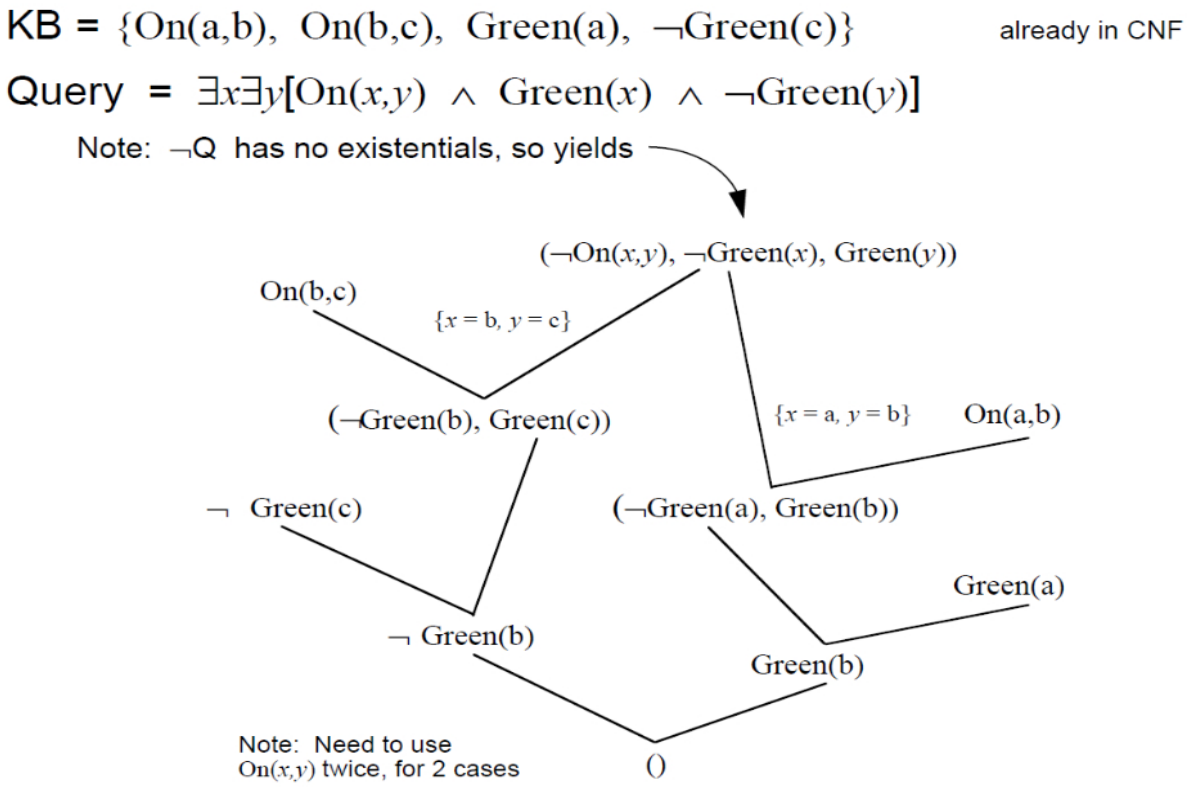
$$(P1, C_1) \wedge \neg P \Rightarrow C_1$$

归结过程示例

示例1：



示例2：



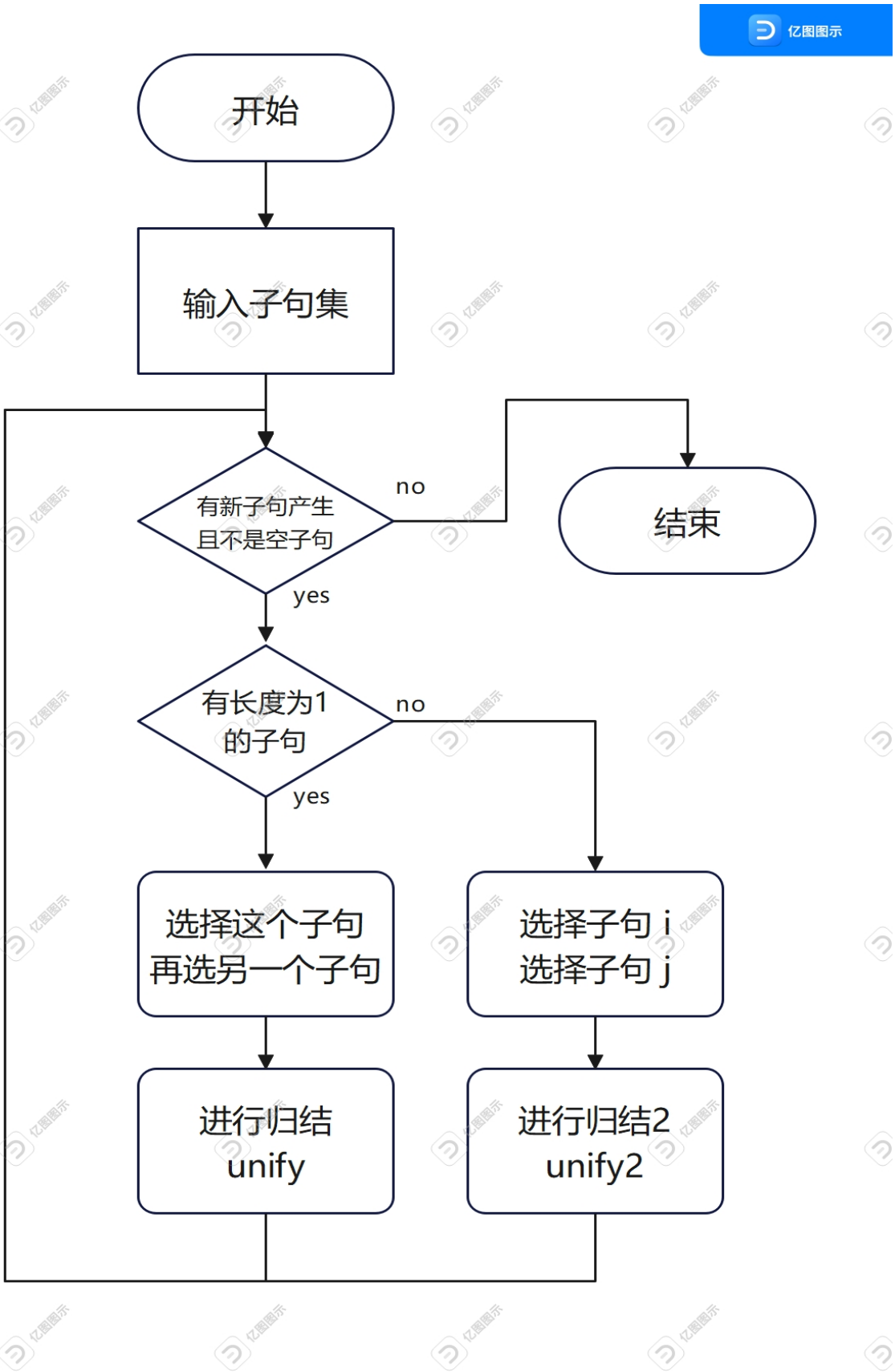
合一算法

1. 如果两个符号是**相等的常量或变量**，则它们是可合一的。如果它们不相等，则它们是不可合一的。
2. 如果一个符号是**一个变量**，而另一个符号是**一个常量或变量**，则将变量绑定到常量或变量上，使它们相等。这个过程被称为**变量绑定**。
3. 如果两个符号**都是函数**，且它们具有相同的函数名和相同数量的参数，则递归地比较它们的每个参数。如果每个参数都可合一，则两个函数可以合一，否则它们是不可合一的。
4. 如果两个符号**都是复合符号**，例如一个带括号的逻辑表达式，那么将它们拆分成它们的子项，**递归**地应用上述步骤进行比较。

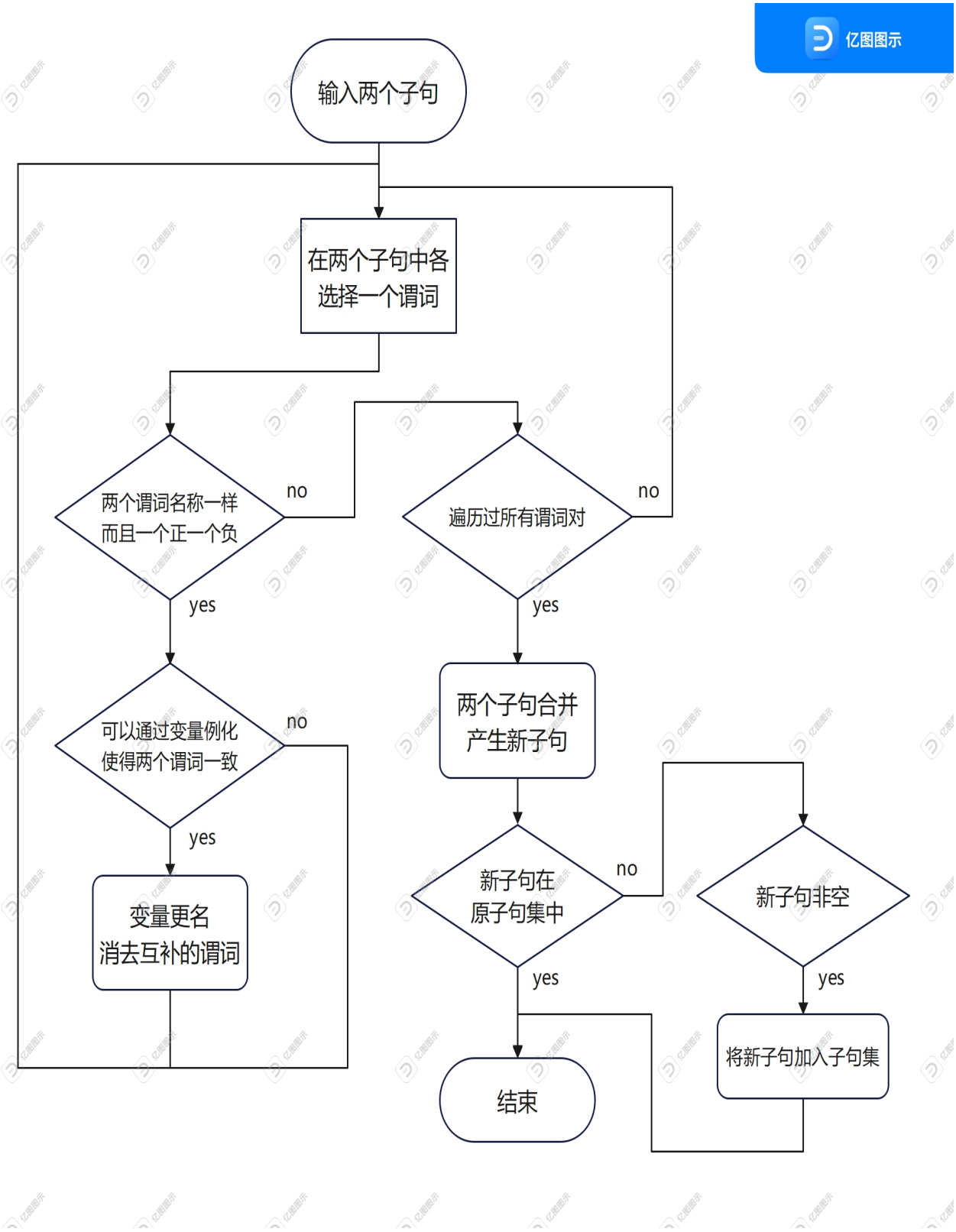
通过这些步骤，合一算法可以找到两个逻辑表达式中的**相等性**

2、流程图

2.1 总体流程图



2.2 单步归结流程图


输入两个子句
在两个子句中各选择一个谓词
两个谓词名称一样而且一个正一个负
no
yes
可以通过变量例化使得两个谓词一致
no
yes
变量更名消去互补的谓词
遍历过所有谓词对
no
yes
两个子句合并产生新子句
新子句在原子句集中
no
yes
新子句非空
yes
将新子句加入子句集
结束

3、关键代码展示（带注释）

```
# 谓词类，存储谓词以及谓词里的各个公式
class WeiCi:
    def __init__(self, input_str):
        self.data = []
        if input_str:
            # 参考input_clause函数，输入str可能第一个元素是','
            if input_str[0] == ',':
                input_str = input_str[1:]
            tmp = ""
            # 以 '(' ',' ')' 将谓词公式分解，data第一项是谓词，后面是谓词里的各个公式
            for item in input_str:
                tmp += item
                # 分隔
                if item == '(' or item == ',' or item == ')':
                    self.data.append(tmp[0:len(tmp)-1])
                    tmp = ""

    # 查看是否为负公式，即有无 "¬" 前缀
    def is_negative(self):
        return self.data[0][0] == "¬"

    # 返回谓词名称
    def get_name(self):
        if self.is_negative():
            return self.data[0][1:]
        else:
            return self.data[0]

    # 返回整个谓词公式
    def get_item(self):
        # tmp = 谓词 + 谓词的左括号
        tmp = self.data[0] + "("
        # 谓词里的公式
        for j in range(1, len(self.data)):
            tmp = tmp + self.data[j]
            if j < len(self.data) - 1:
                tmp = tmp + ","
        tmp += ")"
        return tmp

    # 变量更名
    def rename(self, old_name, new_name):
        for i in range(len(old_name)):
            for j in range(1, len(self.data)):
                if self.data[j] == old_name[i]:
                    self.data[j] = new_name[i]
```

```
# 一阶谓词逻辑归结过程
def solve(clauses):
    flag = True # 标记 继续归结与否
    add_flag = True # 标记 这轮循环有没有生成新子句
    while flag:
        if not add_flag:
            break
        add_flag = False # 循环刚开始，设定该轮循环还没有产生新子句
        for i in range(len(clauses)): # 子句1
            if not flag:
                break
            if len(clauses[i]) == 1: # 子句2
                for j in range(0, len(clauses)):
                    if not flag:
                        break
                    if i == j:
                        continue
                    # 单步归结
                    [ctrl_flag, clauses, tmp_flag] = unify(clauses[i], clauses[j], clauses)
                    # 如果一个归结循环中 归结出一条新子句 则 add_flag 为真
                    add_flag = add_flag or tmp_flag
                    if ctrl_flag == 'continue':
                        continue
                    elif ctrl_flag == 'break':
                        flag = False
                        break
        # 应用 unify1 无法产生新子句， 则应用 unify2
        if not add_flag:
            for i in range(len(clauses)):
                if not flag:
                    break
                for j in range(i+1, len(clauses)): # 子句2
                    if not flag:
                        break
                    # 单步归结
                    [ctrl_flag, clauses, tmp_flag] = unify2(clauses[i], clauses[j], clauses)
                    # 如果一个归结循环中 归结出一条新子句 则 add_flag 为真
                    add_flag = add_flag or tmp_flag
                    if ctrl_flag == 'continue':
                        continue
                    elif ctrl_flag == 'break':
                        flag = False
                        break
        if not add_flag:
            break
    if add_flag:
        print("成功归结出NIT!")
    else:
        print("无新子句产生!")
```

```
# 单步归结过程
def unify2(clause1, clause2, clauses):
    counter = len(clauses) + 1
    # 将自由变量更名为约束变量
    old_name = []
    new_name = []
    pos1 = []
    pos2 = []
    # 在 两个子句 中找相同的谓词，且可以消去，设置pos为其位置
    for i in range(len(clause1)):
        for j in range(len(clause2)):
            # print(i, j)
            # 谓词名字一样，而且是互补项
            if clause1[i].get_name() == clause2[j].get_name() and clause1[i].is_negative() != cl
                pos1.append(i)
                pos2.append(j)
                # 找到可以换名的变量并记录
                for l in range(len(clause2[j].data) - 1):
                    # 是自由变量（设定 只有一个小写字母的公式 为 变量）
                    if len(clause2[j].data[l + 1]) == 1 and len(clause1[i].data[l + 1]) != 1:
                        old_name.append(clause2[j].data[l + 1])
                        new_name.append(clause1[i].data[l + 1])

                    elif len(clause1[i].data[l + 1]) == 1 and len(clause2[j].data[l + 1]) != 1:
                        old_name.append(clause1[j].data[l + 1])
                        new_name.append(clause2[i].data[l + 1])

                    elif clause2[j].data[l + 1] != clause1[i].data[l + 1]:
                        pos1.pop()
                        pos2.pop()
                        break
    # 两个子句无法进行归结
    if not pos1:
        control_flag = 'continue'
        return control_flag, clauses, False
    # 可以归结，改名，消去互补项，生成新子句
    # 记录生成的新子句
    new_clause = []
    # print(pos1, pos2)
    for i in range(len(clause1)):
        # 位置为 pos1 的已经被消去了，所以不在新子句里
        if i not in pos1:
            p = WeiCi("")
            # 往 p 里添加公式
            for item in clause1[i].data:
                p.data.append(item)
            p.rename(old_name, new_name)
            new_clause.append(p)
    for j in range(len(clause2)):
        # 位置为 pos 的已经被消去了，所以不在新子句里
```

```
            if j not in pos2:
                p = WeiCi("")
                # 往 p 里添加公式
                for item in clause2[j].data:
                    p.data.append(item)
                p.rename(old_name, new_name)
                # 避免重复
                if not in_data(p, new_clause):
                    new_clause.append(p)

        pos1 = duplicated_or_not(new_clause, clauses, pos1)
        # 如果生成的子句已存在，跳过加入子句集的过程

        if len(pos1) == 0:
            control_flag = 'continue'
            return control_flag, clauses, False
        # 生成的新的子句加入的子句集中
        clauses.append(new_clause)
        # 展示归结过程

        print(counter, ': R[', clauses.index(clause1)+1, ', ', clauses.index(clause2)+1, '] = ', enc
        # 输出该新子句
        print_clause(new_clause)
        # 判断是否应该结束归结过程
        if end_or_not(new_clause, clauses):
            control_flag = 'break'
            return control_flag, clauses, True
        return 'success', clauses, True
```

4、创新点&优化（如果有）

搜索顺序问题

1. 一开始采用的是**盲目搜索**，归结过程会产生很多无用的新子句，**归结效率非常低。**
2. 于是开始思考：什么样的新子句“更有前途”呢？我认为**更短的新子句更容易归结到空子句。**
3. 于是自然联想到**用两个较短的子句进行归结。**
4. 通过观察实验任务，发现都有 只有一个谓词的子句。用只有一个谓词的子句进行归结，代码比较好写，而且归结出来的子句也“更有前途”。
5. 所以就有了流程图2里面，**优先选择 只有单个谓词的子句** 的判断逻辑。
6. 三个实验任务表明，优先选择 单谓词子句 可以**缩短推理路径**，**更快搜索到空子句。**
7. 后面又产生了新的想法：**利用优先队列来存储子句集**，越短的子句优先级越高，优先取较短的子句进行归结，是不是可以更快呢？但那时代码已经基本成型，改写的话工程量太大，于是仅作讨论，不进行实操。

至于为什么不放图进行比较，这是因为之前的代码已经无了。

多项归结

实现了一次消除两个子句的多个互补谓词

```
1: (A(linux),¬B(windows,x),C(ios))
2: (¬B(windows,x),¬A(linux),¬C(ios))
3: (¬B(macos,x),B(windows,x))
4: (B(windows,x),B(macos,x))
5: R[1, 2] = (¬B(windows,x))
6: R[3, 4] = (B(windows,x))
7: R[5, 6]() = []
成功归结出NIT!

进程已结束,退出代码0
```

三、实验结果与分析

1、实验结果展示示例（可图可表可文字，尽量可视化）

任务一、Aipine Club

```
33: R[2, 11a](w=mike) = (¬C(mike),S(mike))
34: R[3, 6a](x=john) = (S(john),C(john))
35: R[3, 11a](w=john) = (¬C(john),S(john))
36: R[18, 6c](x=tony) = (¬A(tony),S(tony))
37: R[29, 6b](x=mike) = (¬A(mike),C(mike))
38: R[29, 11c](w=mike) = (¬A(mike),¬C(mike))
39: R[29, 14a]() = (C(mike))
40: R[29, 15b]() = (¬C(mike))
41: R[39, 40]() = []
成功归结出NIT!

进程已结束,退出代码0
```

详情见 Lab2\Result\result1

任务二、Graduate Student

```
1: (GradStudent(sue))
2: (¬GradStudent(x),Student(x))
3: (¬Student(x),HardWorker(x))
4: (¬HardWorker(sue))
5: R[1, 2a](x=sue) = (Student(sue))
6: R[4, 3b](x=sue) = (¬Student(sue))
7: R[5, 6]() = []
成功归结出NIT!

进程已结束,退出代码0
```

详情见 Lab2\Result\result2

任务三、Block World

```
1: (On(aa,bb))
2: (On(bb,cc))
3: (Green(aa))
4: (¬Green(cc))
5: (¬On(x,y),¬Green(x),Green(y))
6: R[1, 5a](x=aa, y=bb) = (¬Green(aa),Green(bb))
7: R[2, 5a](x=bb, y=cc) = (¬Green(bb),Green(cc))
8: R[3, 5b](x=aa) = (¬On(aa,y),Green(y))
9: R[3, 6a]() = (Green(bb))
10: R[4, 5c](y=cc) = (¬On(x,cc),¬Green(x))
11: R[4, 7b]() = (¬Green(bb))
12: R[9, 11]() = []
成功归结出NIT!
```

详情见 Lab2\Result\result3

额外内容、多项合一

```
1: (A(linux),¬B(windows,x),C(ios))
2: (¬B(windows,x),¬A(linux),¬C(ios))
3: (¬B(macos,x),B(windows,x))
4: (B(windows,x),B(macos,x))
5: R[1, 2] = (¬B(windows,x))
6: R[3, 4] = (B(windows,x))
7: R[5, 6]() = []
成功归结出NIT!

进程已结束,退出代码0
```

一步消除多个互补的谓词
详情见 Lab2\Result\result4

额外内容、多项合一，无新子句

```
1: (A(linux),¬B(windows,x),C(ios))
2: (¬B(windows,x),¬A(linux),¬C(ios))
3: (B(macos,x),B(windows,x))
4: (¬B(windows,x),B(macos,x))
5: R[1, 2] = (¬B(windows,x))
6: R[3, 4] = (B(macos,x))
无新子句产生!

进程已结束,退出代码0
```

一步消除多个互补的谓词
最后无新子句产生
详情见 Lab2\Result\result5

2、评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

设刚开始有 n 个子句，每个子句最多有 m 个谓词。
$solve$ 函数里对子句进行搜索，用的是**二重循环**
$unify$ 函数里对子句的各个谓词进行搜索，**单层循环**
这就形成了三层循环，再套上一个 $solve$ 函数里的**外循环**，我一开始想着外循环的时间复杂度怎么算呢？难道是算一个 n，然后总体的时间复杂度就是 $O(n \cdot n^2 \cdot m)$
但是想了想又觉得不对，因为**外层循环的次数是不可知的**，可能循环一两次就结束了，也可能因为子句集的增长而循环非常多次。
通过搜索后了解到：一阶谓词逻辑归结算法的时间复杂度可以在**最坏情况下达到指数级别**。这是因为一阶谓词逻辑归结算法需要**枚举所有可能的归结步骤**，而在一些情况下，这些步骤的数量可能会随着公式集合的大小呈指数级增长。
最坏时间复杂度为 $O(2^{n+m})$

四、参考资料

https://blog.csdn.net/starter_____/article/details/88797385