中山大学计算机学院

人工智能

本科生实验报告

（2021 学年春季学期）

课程名称：Artificial Intelligence

教学班级		专业（方向）	
学号		姓名	

一、 实验题目

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),¬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

On(aa,bb)

On(bb,cc)

Green(aa)

¬Green(cc)

(¬On(x,y),¬Green(x),Green(y))

任务二：Graduate Student

GradStudent(sue)

(¬GradStudent(x), Student(x))

(¬Student(x),HardWorker(x))

¬HardWorker(sue)

任务三：Block World

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))

二、 实验内容

1. 算法原理

Clausal form：

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元组，元组中的每一个元素是一个原子公式（或者是原子公式的否定），同时元素之间的关系是析取关系。本次实验处理的输入的子句都是这一形式。

合一算法：

“合一”是指通过变量替换使得两个子句能够被归结，对应的那一组变量替换，它是一种等价的操作。

而“最一般合一”指的是使得两个原子公式等价，最简单的一组变量替换。

我们通过“合一”来让“归结”成为可能。

归结原则：

对于子句集：

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

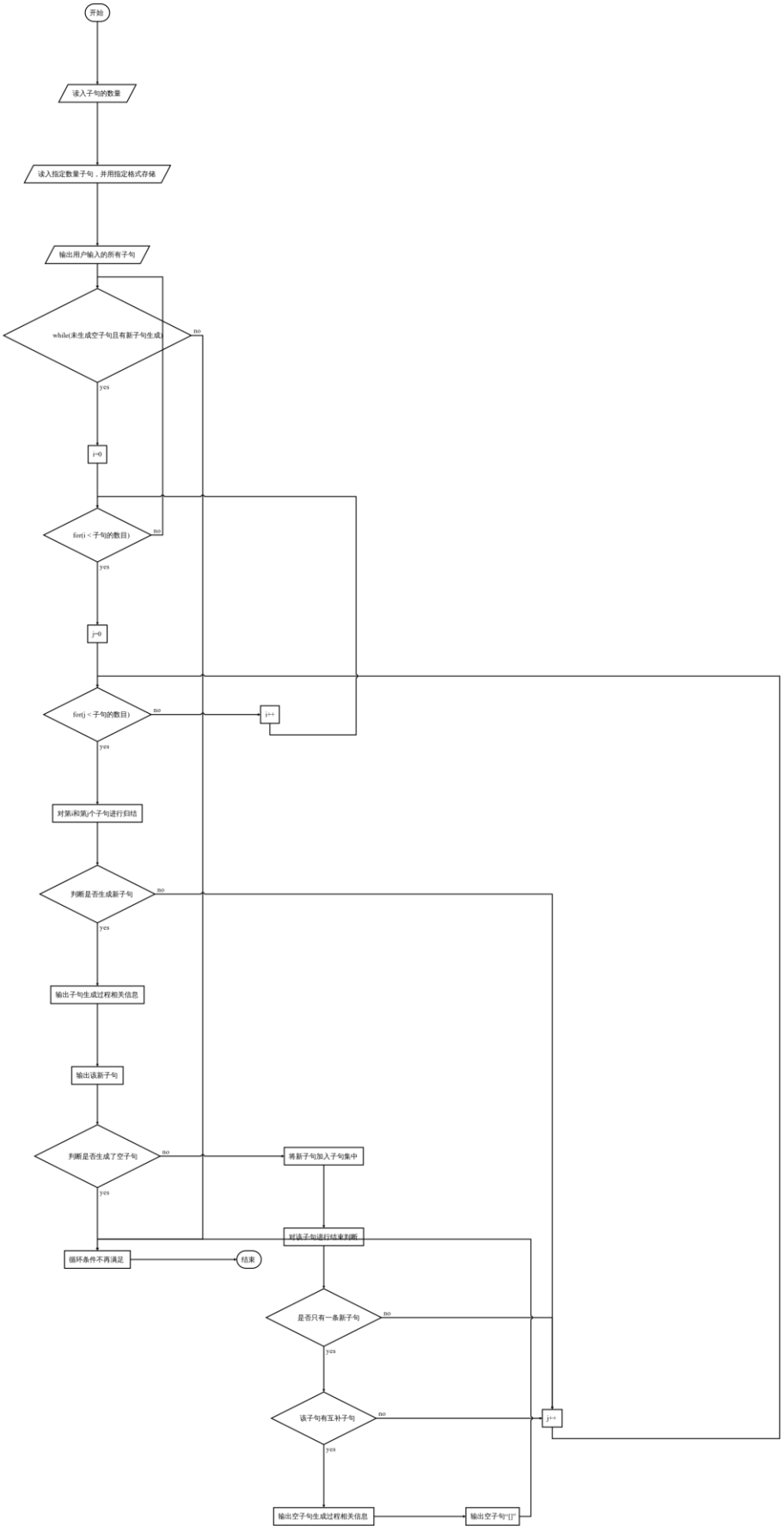
归结原则：

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

它的特殊形式：

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

2. 伪代码/流程图


开始
读入子句的数量
读入指定数量子句，并用指定格式存储
输出用户输入的所有子句
while(未生成空子句且有新子句生成)
no
yes
i=0
for(i < 子句的数目)
no
yes
j=0
for(j < 子句的数目)
no
j++
yes
对第i和第j个子句进行归结
判断是否生成新子句
no
yes
输出子句生成过程相关信息
输出该新子句
判断是否生成了空子句
no
将新子句加入子句集中
yes
对该子句进行结束判断
循环条件不再满足
结束
是否只有一条新子句
no
yes
该子句有互补子句
no
j++
yes
输出空子句生成过程相关信息
输出空子句"[]"

3. 关键代码展示（带注释）

```
class Predicate:  #  我们先定义一个谓词类来对子句中的谓词进行存储
    element = []
    def __init__(self, str_in):
        self.element = []
        if len(str_in) != 0:   #  可能传入字符串为空（这种情况将通过其他方式添加元素）
            if str_in[0] == ',':  #  把原来用于分隔谓词的“,”去掉
                str_in = str_in[1:]
            tmp = ""
            for i in range(len(str_in)): #  将谓词内的元素拆分到列表中
                tmp += str_in[i]
                if str_in[i] == '(' or str_in[i] == ',' or str_in[i] == ')':
                    self.element.append(tmp[0:-1])
                    tmp = ""

    def new(self, list_in):  #  因为只能有一个构造函数，所以此函数用于从列表生成
        for i in range(len(list_in)):   #  谓词拆分成的元素此时存储在列表中
            self.element.append(list_in[i])

    def rename(self, old_name, new_name):  #  用于支持合一 换名的操作，更改谓词中元素
        for i in range(len(old_name)):
            j = 1
            while j < len(self.element):
                if self.element[j] == old_name[i]:
                    self.element[j] = new_name[i]
                j = j + 1

    def get_pre(self):  # 返回谓词的前缀是否为"¬"，是返回布尔值Ture
        return self.element[0][0] == "¬"

    def get_name(self):  # 返回谓词名称（存储在最前面）
        if self.get_pre():
            return self.element[0][1:]
        else:
            return self.element[0]

def print_clause(clause_in):  #从格式化存储的谓词组中还原子句字符串，并打印
    tmp = ""
    if len(clause_in) > 1:
        tmp = tmp + "("
    for i in range(len(clause_in)):
        tmp = tmp + clause_in[i].element[0] + "("
        for j in range(1, len(clause_in[i].element)):
```

```
43.             tmp = tmp + clause_in[i].element[j]
44.             if j < len(clause_in[i].element) - 1:
45.                 tmp = tmp + ","
46.         tmp = tmp + ")"
47.         if i < len(clause_in) - 1:
48.             tmp = tmp + ","
49.     if len(clause_in) > 1:
50.         tmp = tmp + ")"
51.     if(tmp != ""):
52.         print(tmp)
53.
54. def print_msg(key, i, j, old_name, new_name, set_of_clause):  # 输出新子句生成相关信息
55.     tmp = str(len(set_of_clause)) + ": R[" + str(i+1)
56.     if len(new_name) == 0 and len(set_of_clause[i]) != 1:
57.         tmp = tmp + chr(key + 97)
58.     tmp = tmp + ", " + str(j+1) + chr(key + 97) +"]("
59.     for k in range(len(old_name)):
60.         tmp = tmp + old_name[k] + "=" + new_name[k]
61.         if k < len(old_name)-1:
62.             tmp = tmp + ", "
63.     tmp = tmp + ") = "
64.     print(tmp, end="")
65.
66. def end_or_not(new_clause, set_of_clause):
67.     if len(new_clause) == 0:  #  新生成的new_clause 已经为空的情况
68.         print("[]")
69.         return True
70.     if len(new_clause) == 1:  #  查找已有的子句中是否存在与新子句（1 条）互补
71.         for i in range(len(set_of_clause) - 1):  #  set_of_clause[j]超过一个谓词的取或
    的子句
72.             if len(set_of_clause[i]) == 1 and new_clause[0].get_name() == set_of_clau
    se[i][0].get_name() and new_clause[0].element[1:] == set_of_clause[i][0].element[1:]
    and new_clause[0].get_pre() != set_of_clause[i][0].get_pre():
73.                 print(len(set_of_clause) + 1, ": R[", i + 1, ", ", len(set_of_clause)
    , "]() = []", sep="")
74.                 return True
75.     return False  #  不符合条件不结束while 循环
76.
77. def main():  #  包含子句的输入、while 循环和for 循环
78.     set_of_clause = []  #  用于存储子句
79.     print("首先，请输入子句数量：")
80.     num_of_clause = input()
81.     print("下面，请输入",num_of_clause,"条子句：")
82.     for i in range(int(num_of_clause)):  #  获取若干条子句的输入
```

```
        clause_in = input()
        if clause_in == "":
            print("输入过程有误（输入为空），程序将退出")
            return
        if clause_in[0] == '(':  # 如果子句最左侧有括号，则去掉
            clause_in = clause_in[1:-1]
        clause_in = clause_in.replace(' ', '')  # 如果子句内有空格，则去掉
        set_of_clause.append([])  # 一个列表，将输入的子句拆分存储
        tmp = ""  # 用于拆分子句使用的中间变量
        for j in range(len(clause_in)):  # 拆分存储在列表里
            tmp += clause_in[j]
            if clause_in[j] == ')':  # 用')'作为结尾分割成多个谓词公式
                if j + 1 != num_of_clause:
                    clause_tmp = Predicate(tmp)  # 创造一个谓词公式类Predicate 的变量
                    set_of_clause[i].append(clause_tmp)  # 加入到子句集的第 i 个子句中
                tmp = ""

    for i in range(len(set_of_clause)):  # 先输出刚刚输入的子句集里的子句
        print_clause(set_of_clause[i])

    status = True  # 状态为True 时 while 循环会被执行
    while status:
        for i in range(len(set_of_clause)):
            if not status:  # 检查状态
                break
            if len(set_of_clause[i]) == 1:# 只对一个谓词的子句set_of_clause[i]进行下面处理
                for j in range(0, len(set_of_clause)):  #  和其它的子句进行比较
                    if not status:  # 检查状态
                        break
                    if i == j:  #  不和自己比较
                        continue
                    old_name = []
                    new_name = []  #  将自由变量转换为约束变量
                    key = -1  # -1 表示该子句的同名谓词不能进行消去
                    for k in range(len(set_of_clause[j])):  # 在子句set_of_clause[j]中找相同的谓词，且可以消去，设置key 为其位置
                        if set_of_clause[i][0].get_name() == set_of_clause[j][k].get_name() and set_of_clause[i][0].get_pre() != set_of_clause[j][k].get_pre():
                            key = k
                            for l in range(len(set_of_clause[j][k].element) - 1):  # 找到可以换名的变量并记录
                                if len(set_of_clause[j][k].element[l + 1]) == 1:  # 是自由变量
```

```
                                    old_name.append(set_of_clause[j][k].element[l + 1
    ])
                                    new_name.append(set_of_clause[i][0].element[l + 1
    ])
                                elif len(set_of_clause[i][0].element[l + 1]) == 1:
                                    old_name.append(set_of_clause[i][k].element[l + 1
    ])
                                    new_name.append(set_of_clause[j][0].element[l + 1
    ])
                                elif set_of_clause[j][k].element[l + 1] != set_of_cla
    use[i][0].element[l + 1]:
                                    key = -1
                                    break
                            break
                    if key == -1:  #  否则换名 消去 生成新子句
                        continue
                    new_clause = []  # 记录生成的新子句
                    for k in range(len(set_of_clause[j])):
                        if k != key:  # 位置为key 的已经被消去了，所以不在新子句里
                            p = Predicate("")
                            p.new(set_of_clause[j][k].element)
                            p.rename(old_name, new_name)
                            new_clause.append(p)
                    if len(new_clause) == 1:  # 判断是否生成的子句是否与已有重复（不判断
    是否生成了子句）
                        for k in range(len(set_of_clause)):
                            if len(set_of_clause[k]) == 1 and new_clause[0].element =
    = set_of_clause[k][0].element:
                                key = -1
                                break
                    if key == -1:  #  如果生成的子句已存在，跳过加入子句集的过程
                        continue
                    set_of_clause.append(new_clause)  # 生成的新的子句加入的子句集中
                    print_msg(key, i, j, old_name, new_name, set_of_clause)  # 输出生
    成新子句的相关信息
                    print_clause(new_clause)  # 输出该新子句
                    if end_or_not(new_clause, set_of_clause):  #  判断是否应该结束归结
    过程
                        status = False
                        break

    print("Success!")
if __name__ == '__main__':
    main()
```

4. 创新点&优化

上面的代码几乎完美的通过了本次实验的三个用例，但是它真的没有问题吗？

答案是否定的。我们其实只用到了 (A)and(¬A,B,C,...) => (B,C,...) 的规则，而恰巧三个用例都能用这条规则去解决。我们考虑一种需要使用 (¬A,B,C,...)and(A,B,C,...) => (B,C,...) 的规则的情况：

```
1. (A(dear), ¬B(lihua, x))
2. (¬A(dear), ¬B(lihua, x))
3. (B(lihua, x), ¬B(mike, x))
4. (B(lihua, x), B(mike, x))
```

这完全是可以归结的，但是我们的程序却傻眼了，为此，我们需要略做一些升级——对于上面第 108 行的 if 语句，我们在上面 153 行出平行地加入下列代码：

```
1.              else:   # set_of_clause[i]是有多个谓词的子句
2.                  for j in range(0, len(set_of_clause)):  # 找可使用规
    则 (¬A,B,C,...)and(A,B,C,...) => (B,C,...) 的子句 set_of_clause(j)
3.                      key = -1
4.                      if i != j and len(set_of_clause[i]) == len(set_of_clause[j]):
5.                          for k in range(len(set_of_clause[i])):
6.                              if set_of_clause[i][k].element == set_of_clause[j][k].ele
    ment:
7.                                  # 实际情况中，应该进一步考虑各种可进行变量换名的情况
8.                                  continue
9.                              elif set_of_clause[i][k].get_name() == set_of_clause[j][k
    ].get_name() and set_of_clause[i][k].element[1:] == set_of_clause[j][k].element[1:]:
10.                                 # 需要在这里判断变量换名的情况
11.                                 if key != -1:  # 表明已经存在一处不等的情况，无法使用该规
    则进行消除
12.                                     key = -1
13.                                     break
14.                                 key = k
15.                             else:
16.                                 key = -1
17.                                 break
18.                     if key == -1:
19.                         continue
20.                     new_clause = []
21.                     for k in range(len(set_of_clause[i])):
22.                         if k != key:
23.                             p = Predicate("")
24.                             p.new(set_of_clause[j][k].element)
25.                             new_clause.append(p)
26.                     if len(new_clause) == 1:  # 判断是否生成的子句是否与已有重复（不判断
    是否生成了子句）
27.                         for k in range(len(set_of_clause)):
```

```
                            if len(set_of_clause[k]) == 1 and new_clause[0].element ==
    = set_of_clause[k][0].element:
                                key = -1
                                break
                    if key == -1:  #  如果生成的子句已存在，跳过加入子句集的过程
                        continue
                    set_of_clause.append(new_clause)
                    print_msg(key, i, j, [], [], set_of_clause)  # 输出生成新子句的相关
    信息
                    print_clause(new_clause)  # 输出该新子句
                    if end_or_not(new_clause, set_of_clause):  #  判断是否应该结束归结
    过程
                        status = False
                        break
```

这时我们再次尝试处理刚刚我们提到的情况，就可以解决了。我们很容易发现，我们刚添加的部分有一定的代码重用，这部分是可以继续优化的。

另一个优化的点在于是否结束归结过程的判断。我们很容易想到，只要最后归结到一个空的子句，一切就结束了。但是我们的程序里，如果在子句集中已经出现了互补的情况，我们仍然需要走一些弯路才能到达归结到空子句的情况。在上面的代码的第 70-74 行，我们在判断是否结束归结过程时，额外引入了查找是否存在与新生成的子句互补的子句的步骤，从而能够起到提前结束归结过程的效果。

三、 实验结果及分析

1. 实验结果展示示例

我们对是否结束归结过程的判断过程进行优化，能够使得归结的步骤变得更为简洁，优化效果如下对比：

任务一：Aipine Club

（请见下页）

优化前	优化后

```
首先，请输入子句数量：
5
下面，请输入 5 条子句：
On(aa,bb)
On(bb,cc)
Green(aa)
¬Green(cc)
(¬On(x,y),¬Green(x),Green(y))
On(aa,bb)
On(bb,cc)
Green(aa)
¬Green(cc)
(¬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[4, 8b](y=cc) = ¬On(aa,cc)
13: R[1, 5a](x=aa, y=bb) = (¬Green(aa),Green(bb))
14: R[2, 5a](x=bb, y=cc) = (¬Green(bb),Green(cc))
15: R[3, 5b](x=aa) = (¬On(aa,y),Green(y))
16: R[4, 5c](y=cc) = (¬On(x,cc),¬Green(x))
17: R[9, 5b](x=bb) = (¬On(bb,y),Green(y))
18: R[9, 7a]() = Green(cc)
19: R[9, 10b](x=bb) = ¬On(bb,cc)
20: R[9, 11a]() = []
Success!
```

```
首先，请输入子句数量：
5
下面，请输入 5 条子句：
On(aa,bb)
On(bb,cc)
Green(aa)
¬Green(cc)
(¬On(x,y),¬Green(x),Green(y))
On(aa,bb)
On(bb,cc)
Green(aa)
¬Green(cc)
(¬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]() = []
Success!
```

任务二：Graduate Student

优化前	优化后

```
首先，请输入子句数量：
4
下面，请输入 4 条子句：
GradStudent(sue)
(¬GradStudent(x), Student(x))
(¬Student(x),HardWorker(x))
¬HardWorker(sue)
GradStudent(sue)
(¬GradStudent(x),Student(x))
(¬Student(x),HardWorker(x))
¬HardWorker(sue)
5: R[1, 2a](x=sue) = Student(sue)
6: R[4, 3b](x=sue) = ¬Student(sue)
7: R[5, 3a](x=sue) = HardWorker(sue)
8: R[5, 6a]() = []
Success!
```

```
首先，请输入子句数量：
4
下面，请输入 4 条子句：
GradStudent(sue)
(¬GradStudent(x), Student(x))
(¬Student(x),HardWorker(x))
¬HardWorker(sue)
GradStudent(sue)
(¬GradStudent(x),Student(x))
(¬Student(x),HardWorker(x))
¬HardWorker(sue)
5: R[1, 2a](x=sue) = Student(sue)
6: R[4, 3b](x=sue) = ¬Student(sue)
7: R[5, 6]() = []
Success!
```

任务三：Block World

（请见下页）

优化前	优化后

```
首先，请输入子句数量：
11
下面，请输入 11 条子句：
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))
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))
12: R[1, 6a](x=tony) = (S(tony),C(tony))
13: R[1, 11a](w=tony) = (¬C(tony),S(tony))
14: R[2, 6a](x=mike) = (S(mike),C(mike))
15: R[2, 11a](w=mike) = (¬C(mike),S(mike))
16: R[3, 6a](x=john) = (S(john),C(john))
17: R[3, 11a](w=john) = (¬C(john),S(john))
18: R[4, 7b](y=tony) = ¬C(tony)
19: R[4, 9a](u=rain) = ¬L(mike,rain)
20: R[5, 9a](u=snow) = ¬L(mike,snow)
21: R[1, 6a](x=tony) = (S(tony),C(tony))
22: R[1, 11a](w=tony) = (¬C(tony),S(tony))
23: R[2, 6a](x=mike) = (S(mike),C(mike))
24: R[2, 11a](w=mike) = (¬C(mike),S(mike))
25: R[3, 6a](x=john) = (S(john),C(john))
26: R[3, 11a](w=john) = (¬C(john),S(john))
27: R[18, 6c](x=tony) = (¬A(tony),S(tony))
28: R[18, 12b]() = S(tony)
29: R[20, 8a](z=mike) = ¬S(mike)
30: R[1, 6a](x=tony) = (S(tony),C(tony))
31: R[1, 11a](w=tony) = (¬C(tony),S(tony))
32: R[2, 6a](x=mike) = (S(mike),C(mike))
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[1, 6a](x=tony) = (S(tony),C(tony))
42: R[1, 11a](w=tony) = (¬C(tony),S(tony))
```

```
首先，请输入子句数量：
11
下面，请输入 11 条子句：
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))
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))
12: R[1, 6a](x=tony) = (S(tony),C(tony))
13: R[1, 11a](w=tony) = (¬C(tony),S(tony))
14: R[2, 6a](x=mike) = (S(mike),C(mike))
15: R[2, 11a](w=mike) = (¬C(mike),S(mike))
16: R[3, 6a](x=john) = (S(john),C(john))
17: R[3, 11a](w=john) = (¬C(john),S(john))
18: R[4, 7b](y=tony) = ¬C(tony)
19: R[4, 9a](u=rain) = ¬L(mike,rain)
20: R[5, 9a](u=snow) = ¬L(mike,snow)
21: R[1, 6a](x=tony) = (S(tony),C(tony))
22: R[1, 11a](w=tony) = (¬C(tony),S(tony))
23: R[2, 6a](x=mike) = (S(mike),C(mike))
24: R[2, 11a](w=mike) = (¬C(mike),S(mike))
25: R[3, 6a](x=john) = (S(john),C(john))
26: R[3, 11a](w=john) = (¬C(john),S(john))
27: R[18, 6c](x=tony) = (¬A(tony),S(tony))
28: R[18, 12b]() = S(tony)
29: R[20, 8a](z=mike) = ¬S(mike)
30: R[1, 6a](x=tony) = (S(tony),C(tony))
31: R[1, 11a](w=tony) = (¬C(tony),S(tony))
32: R[2, 6a](x=mike) = (S(mike),C(mike))
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]() = []
Success!
```

```
43: R[2, 6a](x=mike) = (S(mike),C(mike))
44: R[2, 11a](w=mike) = (¬C(mike),S(mike))
45: R[3, 6a](x=john) = (S(john),C(john))
46: R[3, 11a](w=john) = (¬C(john),S(john))
47: R[18, 6c](x=tony) = (¬A(tony),S(tony))
48: R[29, 6b](x=mike) = (¬A(mike),C(mike))
49: R[29, 11c](w=mike) = (¬A(mike),¬C(mike))
50: R[39, 11b](w=mike) = (¬A(mike),S(mike))
51: R[39, 15a]() = S(mike)
52: R[39, 38b]() = ¬A(mike)
53: R[39, 40a]() = []
Success!
```

特殊举例：

这里我们把对优化的另一个点的对比也放在下面：

优化前	优化后
首先，请输入子句数量： 4 下面，请输入 4 条子句： (A(dear), ¬B(lihua, x)) (¬A(dear), ¬B(lihua, x)) (B(lihua, x), ¬B(mike, x)) (B(lihua, x), B(mike, x)) (A(dear),¬B(lihua,x)) (¬A(dear),¬B(lihua,x)) (B(lihua,x),¬B(mike,x)) (B(lihua,x),B(mike,x)) 程序陷入死循环，无法完成归结过程，同时电脑风扇起飞。	首先，请输入子句数量： 4 下面，请输入 4 条子句： (A(dear), ¬B(lihua, x)) (¬A(dear), ¬B(lihua, x)) (B(lihua, x), ¬B(mike, x)) (B(lihua, x), B(mike, x)) (A(dear),¬B(lihua,x)) (¬A(dear),¬B(lihua,x)) (B(lihua,x),¬B(mike,x)) (B(lihua,x),B(mike,x)) 5: R[1a, 2a]() = ¬B(lihua,x) 6: R[3b, 4b]() = B(lihua,x) 7: R[5, 6]() = [] Success! 可以正常完成归结过程

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）

我们主要有两项评价指标：一是是否能够正常完成归结过程；二是完成归结过程所需要的步骤数量。

根据上面我们的举例，在第一项指标中，优化前的程序不能对部分情况下的子句集完成归结过程，也就是不能够处理 (A)and(¬A,B,C,...) => (B,C,...) 的规则不能处理的情况，而优化后的程序将 (¬A,B,C,...)and(A,B,C,...) => (B,C,...) 的规则引入了，从而解决了问题。

在第二项指标中，对于优化后的程序，由于在程序逻辑上提前处理了可以生成空子句的情况，所以在步骤上更少，表现更优

四、 参考资料

《人工智能》（第三版）_贲可荣、张彦铎_清华大学出版社