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 2 班**	专业（方向）	**计算机科学与技术**
学号	**22336126**	姓名	**李漾**

一、 实验题目

一阶逻辑的归结推理

二、 实验内容

1. 算法原理

（1）输入：给定一组一阶逻辑公式（子句），以及待证明的目标公式（子句）。

（2）转化为合取范式：通过应用否定范式（Negation Normal Form, NNF）和 Skolemization 等技术，将所有输入公式转化为合取范式（Conjunctive Normal Form, CNF），确保每个公式都是一个子句（clause），即一个谓词的合取或者多个谓词的析取。

（3）构建初始子句集合：将所有合取范式的子句放入一个初始的子句集合中。

（4）循环应用归结规则：

- 在每次迭代中，选择两个子句进行归结。这两个子句中一个含有某个谓词的肯定形式，另一个含有该谓词的否定形式。
- 对于选定的两个子句，通过匹配变量和重命名等方式找到归结合一致的部分，将它们从两个子句中删除，合并剩余的部分。
- 如果得到的新子句不为空且不是之前已经推导出的子句，将其加入到子句集合中。

继续循环，直到无法再应用归结规则为止。

（5）检查是否证明成功：如果得到空子句，则证明成功；如果无法再应用归结规则且仍未得到空子句，则证明失败。

2. 伪代码

（1）谓词类

```
函数 __init__(self, str_in):

    elements = 空列表
```

```
    如果 str_in 的长度不为 0：
        如果 str_in 的第一个字符是 ','：
            将 str_in 的第一个字符去掉
        临时变量 tmp 设为 空字符串
        对于 i 在范围(0, len(str_in)) 中：
            将 str_in 的第 i 个字符添加到 tmp 中
            如果 str_in 的第 i 个字符在 ['(', ',', ')'] 中：
                将 tmp 中去掉最后一个字符后的部分添加到 elements 列表中
                重置 tmp 为空字符串函数 __init__(self, str_in):
    elements = 空列表
    如果 str_in 的长度不为 0：
        如果 str_in 的第一个字符是 ','：
            将 str_in 的第一个字符去掉
        临时变量 tmp 设为 空字符串
        对于 i 在范围(0, len(str_in)) 中：
            将 str_in 的第 i 个字符添加到 tmp 中
            如果 str_in 的第 i 个字符在 ['(', ',', ')'] 中：
                将 tmp 中去掉最后一个字符后的部分添加到 elements 列表中
                重置 tmp 为空字符串
函数 new(self, list_in):
    对于 element 在 list_in 中：
        将 element 添加到 self.elements 中
函数 get_pre(self):
    返回 self.elements[0] 的第一个字符是否为 "~"
函数 get_name(self):
    如果 self.get_pre() 为真：
        返回 self.elements[0] 的第二个字符到末尾
```

```
        否则：
            返回 self.elements[0]
```

（2）子句转换为字符串

```
    函数 print_clause(clause_in):
        clause_str 设为空字符串
        如果 clause_in 的长度大于 1：
            将 "(" 添加到 clause_str 中
        对于 i 在范围(0, len(clause_in)) 中：
            将 clause_in[i].elements[0] 添加到 clause_str 中
            如果 clause_in[i].elements 的长度大于 1：
                将 "(" 添加到 clause_str 中
                对于 j 在范围(1, len(clause_in[i].elements)) 中：
                    将 clause_in[i].elements[j] 添加到 clause_str 中
                    如果 j 小于 clause_in[i].elements 的长度减 1：
                        将 "," 添加到 clause_str 中
                将 ")" 添加到 clause_str 中
            如果 i 小于 clause_in 的长度减 1：
                将 "," 添加到 clause_str 中
        如果 clause_in 的长度大于 1：
            将 ")" 添加到 clause_str 中
        返回 clause_str
```

（3）描述归结推理过程中的变换

```
    函数 print_msg(key, i, j, old_name, new_name):
        msg 设为 len(set_of_clause) 的字符串形式 + ": R[" + (i + 1) 的字符串形式
        如果 new_name 的长度为 0 并且 set_of_clause[i] 的长度不为 1：
            将 chr(key + 97) 添加到 msg 中
        将 ", " + (j + 1) 的字符串形式 + chr(key + 97) 添加到 msg 中
```

```
        将 "](" 添加到 msg 中
        对于 k 在范围(0, len(old_name)) 中：
            将 old_name[k] + "=" + new_name[k] 添加到 msg 中
            如果 k 小于 old_name 的长度减 1：
                将 ", " 添加到 msg 中
        将 ") = " 添加到 msg 中
        返回 msg
```

（4）检查推理过程是否结束

```
    函数 end_or_not():
        对于 new_clause 在 set_of_clause 中：
            如果 new_clause 为空：
                返回 真
        返回 假
```

（5）主函数

```
    函数 main():
        将 set_of_clause 设为全局变量并初始化为空列表
        将 clauses 设为输入的谓词逻辑子句列表
        对于 clause_in 在 clauses 中：
            如果 clause_in 的第一个字符是 "(":
                将 clause_in 的第一个字符去掉
            将 clause_in 中的空格去掉
            将一个空列表添加到 set_of_clause 中
            临时变量 tmp 设为 空字符串
            对于 j 在范围(0, len(clause_in)) 中：
                将 clause_in 的第 j 个字符添加到 tmp 中
                如果 clause_in 的第 j 个字符是 ")":
                    将 tmp 构建为一个谓词对象 clause_tmp
                    将 clause_tmp 添加到 set_of_clause 的最后一个子句中
```

```
                    重置 tmp 为空字符串


        对于 i 在范围(0, len(set_of_clause)) 中：
            打印 print_clause(set_of_clause[i])


        将 status 设为 真
        当 status 为 真 时：
            对于 i 在范围(0, len(set_of_clause)) 中：
                如果 status 为 假：
                    跳出循环
                如果 set_of_clause[i] 的长度为 1：
                    对于 j 在范围(0, len(set_of_clause)) 中：
                        如果 status 为 假：
                            跳出循环
                        如果 i 等于 j：
                            继续下一次循环
                        将 old_name 设为 空列表
                        将 new_name 设为 空列表
                        将 key 设为 -1
                        对于 k 在范围(0, len(set_of_clause[j])) 中：
                            如果 set_of_clause[i][0].get_name() 等于
set_of_clause[j][k].get_name() 并且 set_of_clause[i][0].get_pre() 不
等于 set_of_clause[j][k].get_pre()：
                                将 key 设为 k
                                对于 l 在范围(1,
len(set_of_clause[j][k].elements)) 中：
                                    如果 set_of_clause[j][k].elements[l]
的长度为 1：
```

```
                                        将
set_of_clause[j][k].elements[l] 添加到 old_name 中

                                        将
set_of_clause[i][0].elements[l] 添加到 new_name 中

                                    否则如果
set_of_clause[i][0].elements[l] 的长度为 1：

                                        将
set_of_clause[i][0].elements[l] 添加到 old_name 中

                                        将
set_of_clause[j][k].elements[l] 添加到 new_name 中

                                    否则如果
set_of_clause[j][k].elements[l] 不等于
set_of_clause[i][0].elements[l]：

                                        将 key 设为 -1

                                        跳出循环

                                跳出循环

                        如果 key 等于 -1：

                            继续下一次循环

                        将 new_clause 设为 空列表

                        对于 k 在范围(0, len(set_of_clause[j])) 中：

                            如果 k 不等于 key：

                                将 p 设为 新的谓词对象，参数为空字符串

                                谓词对象 p 执行 new 操作，参数为
set_of_clause[j][k].elements

                                谓词对象 p 执行 rename 操作，参数为
old_name 和 new_name

                                将 p 添加到 new_clause 中

                        如果 new_clause 的长度为 1：

                            对于 k 在范围(0, len(set_of_clause)) 中：

                                如果 set_of_clause[k] 的长度为 1 并且
new_clause[0].elements 等于 set_of_clause[k][0].elements：
```

```
                                    将 key 设为 -1
                                    跳出循环
                            如果 key 等于 -1：
                                继续下一次循环
                            将 new_clause 添加到 set_of_clause 中
                            打印 print_msg(key, i, j, old_name, new_name)，末尾不换行
                            打印 print_clause(new_clause)
                            如果 end_or_not() 为 真：
                                将 status 设为 假
                                跳出循环
                    否则：
                        对于 j 在范围(0, len(set_of_clause)) 中：
                            将 key 设为 -1
                            如果 i 不等于 j 并且 set_of_clause[i] 的长度等于 set_of_clause[j] 的长度：
                                对于 k 在范围(0, len(set_of_clause[i])) 中：
                                    如果 set_of_clause[i][k].elements 不等于 set_of_clause[j][k].elements：
                                        继续下一次循环
                                    否则如果 set_of_clause[i][k].get_name() 等于 set_of_clause[j][k].get_name() 并且 set_of_clause[i][k].elements[1:] 等于 set_of_clause[j][k].elements[1:]：
                                        如果 key 不等于 -1：
                                            将 key 设为 -1
                                            跳出循环
                                        将 key 设为 k
                                    否则：
```

```
                                    将 key 设为 -1
                                    跳出循环
                            如果 key 等于 -1：
                                继续下一次循环
                            将 new_clause 设为 空列表
                            对于 k 在范围(0, len(set_of_clause[i])) 中：
                                如果 k 不等于 key：
                                    将 p 设为 新的谓词对象，参数为空字符串
                                    谓词对象 p 执行 new 操作，参数为
set_of_clause[j][k].elements
                                    将 p 添加到 new_clause 中
                            如果 new_clause 的长度为 1：
                                对于 k 在范围(0, len(set_of_clause)) 中：
                                    如果 set_of_clause[k] 的长度为 1 并且
new_clause[0].elements 等于 set_of_clause[k][0].elements：
                                        将 key 设为 -1
                                        跳出循环
                            如果 key 等于 -1：
                                继续下一次循环
                            将 new_clause 添加到 set_of_clause 中
                            打印 print_msg(key, i, j, [], [])，末尾不换行
                            打印 print_clause(new_clause)
                            如果 end_or_not() 为 真：
                                将 status 设为 假
                                跳出循环
                    如果 status 为 真：
                        跳出循环
```

```
        打印 "Success!"
```

3. 关键代码展示（带注释）

```
class Predicate:
    def __init__(self, str_in):
        self.elements = []    # 谓词对象的元素列表
        if len(str_in) != 0:
            if str_in[0] == ',':
                str_in = str_in[1:]
            tmp = ""
            for i in range(len(str_in)):
                tmp += str_in[i]
                if str_in[i] in ['(', ',', ')']:
                    self.elements.append(tmp[0:-1])    # 将元素添加到谓词对象的元素列表中
                    tmp = ""

    def new(self, list_in):
        for element in list_in:
            self.elements.append(element)    # 添加新元素到谓词对象的元素列表中

    def rename(self, old_name, new_name):
        for i in range(len(self.elements)):
            for j in range(len(old_name)):
                if self.elements[i] == old_name[j]:
                    self.elements[i] = new_name[j]    # 将谓词对象的元素进行重命名

    def get_pre(self):
        return self.elements[0][0] == "~"    # 返回谓词对象是否为否定形式

    def get_name(self):
        if self.get_pre():
            return self.elements[0][1:]
        else:
            return self.elements[0]    # 返回谓词对象的名称


def print_clause(clause_in):
    """
    打印谓词逻辑子句

    参数：
```

```
    clause_in -- 谓词逻辑子句的列表

    返回：
    clause_str -- 表示谓词逻辑子句的字符串
    """
    clause_str = ""
    if len(clause_in) > 1:
        clause_str += "("
    for i in range(len(clause_in)):
        clause_str += clause_in[i].elements[0]
        if len(clause_in[i].elements) > 1:
            clause_str += "("
            for j in range(1, len(clause_in[i].elements)):
                clause_str += clause_in[i].elements[j]
                if j < len(clause_in[i].elements) - 1:
                    clause_str += ","
            clause_str += ")"
        if i < len(clause_in) - 1:
            clause_str += ","
    if len(clause_in) > 1:
        clause_str += ")"
    return clause_str


def print_msg(key, i, j, old_name, new_name):
    """
    打印归结过程的消息

    参数：
    key -- 归结过程的关键字
    i -- 子句集合中的索引
    j -- 子句集合中的索引
    old_name -- 旧的谓词名称
    new_name -- 新的谓词名称

    返回：
    msg -- 归结过程的消息字符串
    """
    msg = str(len(set_of_clause)) + ": R[" + str(i + 1)
    if len(new_name) == 0 and len(set_of_clause[i]) != 1:
        msg += chr(key + 97)
    msg += ", " + str(j + 1) + chr(key + 97) + "]("
    for k in range(len(old_name)):
```

```
            msg += old_name[k] + "=" + new_name[k]
            if k < len(old_name) - 1:
                msg += ", "
    msg += ") = "
    return msg


def end_or_not():
    """
    判断推理是否结束

    返回：
    boolean --  推理是否结束的布尔值
    """
    for new_clause in set_of_clause:
        if not new_clause:
            return True
    return False


def main():
    global set_of_clause
    set_of_clause = []    #  子句集合初始化为空列表
    clauses = [
        #此处为所需要归结的子句集
    ]

    for clause_in in clauses:
        if clause_in[0] == '(':
            clause_in = clause_in[1:-1]
        clause_in = clause_in.replace(' ', '')
        set_of_clause.append([])
        tmp = ""
        for j in range(len(clause_in)):
            tmp += clause_in[j]
            if clause_in[j] == ')':
                clause_tmp = Predicate(tmp)
                set_of_clause[-1].append(clause_tmp)
                tmp = ""

    for i in range(len(set_of_clause)):
        print(print_clause(set_of_clause[i]))    #  打印初始的子句
```

```
    status = True
    while status:
        for i in range(len(set_of_clause)):
            if not status:
                break
            if len(set_of_clause[i]) == 1:
                for j in range(len(set_of_clause)):
                    if not status:
                        break
                    if i == j:
                        continue
                    old_name = []
                    new_name = []
                    key = -1
                    for k in range(len(set_of_clause[j])):
                        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(1, len(set_of_clause[j][k].elements)):
                                if len(set_of_clause[j][k].elements[l]) == 1:
                                    old_name.append(set_of_clause[j][k].elements[l])
                                    new_name.append(set_of_clause[i][0].elements[l])
                                elif len(set_of_clause[i][0].elements[l]) == 1:
                                    old_name.append(set_of_clause[i][0].elements[l])
                                    new_name.append(set_of_clause[j][k].elements[l])
                                elif set_of_clause[j][k].elements[l] !=
set_of_clause[i][0].elements[l]:
                                    key = -1
                                    break
                            break
                    if key == -1:
                        continue
                    new_clause = []
                    for k in range(len(set_of_clause[j])):
                        if k != key:
                            p = Predicate("")
                            p.new(set_of_clause[j][k].elements)
                            p.rename(old_name, new_name)
                            new_clause.append(p)
                    if len(new_clause) == 1:
                        for k in range(len(set_of_clause)):
                            if len(set_of_clause[k]) == 1 and new_clause[0].elements ==
set_of_clause[k][0].elements:
```

```
                                key = -1
                                break
                    if key == -1:
                        continue
                    set_of_clause.append(new_clause)
                    print(print_msg(key, i, j, old_name, new_name), end="")
                    print(print_clause(new_clause))    # 打印归结得到的新子句
                    if end_or_not():
                        status = False
                        break
            else:
                for j in range(len(set_of_clause)):
                    key = -1
                    if i != j and len(set_of_clause[i]) == len(set_of_clause[j]):
                        for k in range(len(set_of_clause[i])):
                            if set_of_clause[i][k].elements == set_of_clause[j][k].elements:
                                continue
                            elif set_of_clause[i][k].get_name() == set_of_clause[j][k].get_name() and set_of_clause[i][k].elements[1:] == set_of_clause[j][k].elements[1:]:
                                if key != -1:
                                    key = -1
                                    break
                                key = k
                            else:
                                key = -1
                                break
                    if key == -1:
                        continue
                    new_clause = []
                    for k in range(len(set_of_clause[i])):
                        if k != key:
                            p = Predicate("")
                            p.new(set_of_clause[j][k].elements)
                            new_clause.append(p)
                    if len(new_clause) == 1:
                        for k in range(len(set_of_clause)):
                            if len(set_of_clause[k]) == 1 and new_clause[0].elements == set_of_clause[k][0].elements:
                                key = -1
                                break
                    if key == -1:
                        continue
```

```
                    set_of_clause.append(new_clause)
                    print(print_msg(key, i, j, [], []), end="")
                    print(print_clause(new_clause))   # 打印归结得到的新子句
                    if end_or_not():
                        status = False
                        break
            if not status:
                break

    print("Success!")


if __name__ == '__main__':
    main()
```

4. 创新点&优化（如果有）

是否结束归结过程的判断。如果是按照我自己平时的做题习惯,只要最后归结到一个空的子句，即发现两个可以形成矛盾的时候，归结就结束了。但是我的程序里，如果在子句集中已经出现了互补的情况，却仍然需要继续进行归结过程才能到达归结到空子句的情况。因此，我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。

修改后的函数为：

```
    def end_or_not(new_clause, set_of_clause):

        if len(new_clause) == 0:

            print("[]")

            return True

        if len(new_clause) == 1:

            for i in range(len(set_of_clause) - 1):

                if len(set_of_clause[i]) == 1 and new_clause[0].get_name()
== set_of_clause[i][0].get_name() and new_clause[0].elements[1:] ==
set_of_clause[i][0].elements[1:] and new_clause[0].get_pre() !=
set_of_clause[i][0].get_pre():

                    print(len(set_of_clause) + 1, ": R[", i + 1, ", ",
len(set_of_clause), "]() = []", sep="")

                    return True

        return False
```

三、 实验结果及分析

1. 实验结果展示示例

Example：

```
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!
```

T1：

```
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),S(w))
12: R[1, 6a](x=tony) = (S(tony),C(tony))
13: R[1, 11a](w=tony) = S(tony)
14: R[2, 6a](x=mike) = (S(mike),C(mike))
15: R[2, 11a](w=mike) = S(mike)
16: R[3, 6a](x=john) = (S(john),C(john))
17: R[3, 11a](w=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[2, 6a](x=mike) = (S(mike),C(mike))
23: R[3, 6a](x=john) = (S(john),C(john))
24: R[15, 8b](z=mike) = L(mike,snow)
25: R[17, 8b](z=john) = L(john,snow)
26: R[18, 6c](x=tony) = (~A(tony),S(tony))
27: R[20, 8a](z=mike) = ~S(mike)
28: R[20, 24a]() =
Success!
```

T2：

```
On(tony,mike)
On(mike,john)
Green(tony)
~Green(john)
(~On(x,y),~Green(x),Green(y))
6: R[1, 5a](x=tony, y=mike) = (~Green(tony),Green(mike))
7: R[2, 5a](x=mike, y=john) = (~Green(mike),Green(john))
8: R[3, 5b](x=tony) = (~On(tony,y),Green(y))
9: R[3, 6a]() = Green(mike)
10: R[4, 5c](y=john) = (~On(x,john),~Green(x))
11: R[4, 7b]() = ~Green(mike)
12: R[9, 11]() = []
Success!
```

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）

因为在进行 T3 的时候出现的归结冗杂非常明显，我才思考到需要进行优化

优化前：

```
On(tony,mike)
On(mike,john)
Green(tony)
~Green(john)
(~On(x,y),~Green(x),Green(y))
6: R[1, 5a](x=tony, y=mike) = (~Green(tony),Green(mike))
7: R[2, 5a](x=mike, y=john) = (~Green(mike),Green(john))
8: R[3, 5b](x=tony) = (~On(tony,y),Green(y))
9: R[3, 6a]() = Green(mike)
10: R[4, 5c](y=john) = (~On(x,john),~Green(x))
11: R[4, 7b]() = ~Green(mike)
12: R[4, 8b](y=john) = ~On(tony,john)
13: R[1, 5a](x=tony, y=mike) = (~Green(tony),Green(mike))
14: R[2, 5a](x=mike, y=john) = (~Green(mike),Green(john))
15: R[3, 5b](x=tony) = (~On(tony,y),Green(y))
16: R[4, 5c](y=john) = (~On(x,john),~Green(x))
17: R[9, 5b](x=mike) = (~On(mike,y),Green(y))
18: R[9, 7a]() = Green(john)
19: R[9, 10b](x=mike) = ~On(mike,john)
20: R[9, 11a]() =
Success!
```

优化后：

```
On(tony,mike)
On(mike,john)
Green(tony)
~Green(john)
(~On(x,y),~Green(x),Green(y))
6: R[1, 5a](x=tony, y=mike) = (~Green(tony),Green(mike))
7: R[2, 5a](x=mike, y=john) = (~Green(mike),Green(john))
8: R[3, 5b](x=tony) = (~On(tony,y),Green(y))
9: R[3, 6a]() = Green(mike)
10: R[4, 5c](y=john) = (~On(x,john),~Green(x))
11: R[4, 7b]() = ~Green(mike)
12: R[9, 11]() = []
Success!
PS D:\人工智能>
```

非常明显提高了运行的效率

五、参考资料

https://blog.csdn.net/yinxian9019/article/details/90216359