> [!note] **中山大学计算机学院-人工智能-本科生实验报告**
> 2023学年春季学期
> 课程名称：Artificial Intelligence

教学班级	计算机科学与技术	专业（方向）	系统结构
学号	22336018	姓名	蔡可豪

1 实验题目

归结原理实验。

2 实验内容

编写程序，实现归结原理（可以适用于一阶逻辑），并且应用于两个例子上进行推理。

同时按照一定的格式输出，方便助教检查。

2.1 算法原理

> *用自己的话简单解释一下对算法和模型的理解*

要让机器学会推理，首先就是要给机器一种非常直接，易于循环迭代的推理规则。

从数学上，最适合机器运行的推理规则就是归结

> 归结的本质是寻找矛盾，从矛盾推导出新的句子，如此往复直到最终的矛盾或者最终不矛盾。

在算法实现的过程中主要有三个部分：

1. 字符串格式处理：将输入转换成便于程序执行的格式
2. 单步归结算法
3. 最一般合一算法

2.2 伪代码

由于代码过长，仅展示部分关键算法的伪代码

MGU

```
函数 MGU(predicate1, predicate2)
    初始化替换方案字典 replace = {}

    对于 i 从 0 到 predicate1.arguments的长度 - 1
        argument1 = predicate1.arguments[i]
        argument2 = predicate2.arguments[i]

        如果 argument1 和 argument2 的长度都为 1
            将 argument1 映射到 argument2 在 replace 中
        否则如果 argument1 长度为 1 且 argument2 长度不为 1
```

```
            将 argument1 映射到 argument2 在 replace 中
        否则如果 argument1 长度不为 1 且 argument2 长度为 1
            将 argument2 映射到 argument1 在 replace 中
        否则如果 argument1 和 argument2 长度都不为 1 且 argument1 不等于 argument2
            返回 False（表示无法合一）

    返回 replace（替换方案字典）
```

Resolve

```
函数 resolve(clause1, clause2, predicate1, predicate2, replace)
    初始化新子句 new_clause = Clause([])

    对于 clause1 中的每个谓词 predicate
        如果 predicate 不等于 predicate1
            将 predicate 的深拷贝添加到 new_clause 的 predicates 中

    对于 clause2 中的每个谓词 predicate
        如果 predicate 不等于 predicate2
            将 predicate 的深拷贝添加到 new_clause 的 predicates 中

    对于 replace 中的每个替换项 (key, value)
        对于 new_clause 中的每个 predicate
            对于 predicate.arguments 的每个索引 i
                如果 predicate.arguments[i] 等于 key
                    将 predicate.arguments[i] 替换为 value

    返回 new_clause

```

Resolution_algorithm(main loop)

```
函数 resolution_algorithm(KB, debug = False)
    初始化访问过的组合列表 visited = []
    初始化目标测试标志 goal_test = False

    循环直到 goal_test 为真 或 无法继续找到匹配项
        初始化 find = False

        对于 KB 中的每个子句索引 i
            对于 i+1 到 KB的长度-1 中的每个子句索引 j
                clause1 = KB[i]
                clause2 = KB[j]

                对于 clause1 的每个谓词 predicate1 和其索引 index1
                    对于 clause2 的每个谓词 predicate2 和其索引 index2
                        signal = 构造标识符(i+1, index1, j+1, index2)

                        如果 predicate1 和 predicate2 可以匹配 且 signal 不在 visited 中
                            replace = MGU(predicate1, predicate2)
                            如果 replace 不为 False
```

```
                                将 signal 添加到 visited 中
                                find = True
                                new_clause = resolve(clause1, clause2, predicate1,
predicate2, replace)

                                如果 debug 为真
                                    调用 debug_info(KB)
                                调用 display_info 显示信息

                                如果 new_clause 为空
                                    设置 goal_test 为真 并 跳出循环

                                将 new_clause 添加到 KB 中

                            如果找到匹配
                                跳出循环(==指的是跳出所有的循环==)

    输出结果

```

2.3 关键代码展示（含注释）

首先定义了一个`predicate`类，类有名字，参数列表和否的判断。

同时支持了一个`print_predicate`函数，用于向用户展示人能看的懂的谓词语法。

除此之外，有一个`deepcopy`方法，这里折磨了我好久。

```python
class Predicate:
    def __init__(self, name, arguments, negated=False):
        self.name = name
        self.arguments = arguments
        self.negated = negated
    def print_predicate(self):
        """Print the predicate in a readable format."""
        output = ""
        if self.negated:
            output += "¬"
        output += self.name + "("
        for argument in self.arguments:
            output += argument
            if argument is not self.arguments[-1]:
                output += ", "
        output += ")"
        return output
    def deepcopy(self):
        """Create a deep copy of the Predicate instance."""
        return Predicate(copy.deepcopy(self.name),
                         copy.deepcopy(self.arguments),
                         copy.deepcopy(self.negated))
```

再然后是一个`clause`类，主要的内容就是有一个predicates列表，因为一个语句可能包含多个predicate。

同样，有print函数和deepcopy函数。

```
class Clause:
    def __init__(self, predicates = []):
        self.predicates = predicates # It is a LIST of predicate above

    def print_clause(self, direct = True):
        """Print the clause in a readable format. If direct is False, return the string instead of printing it."""
        if len(self.predicates) == 1:
            if direct is not False:
                print(self.predicates[0].print_predicate())
            return
        output = "("
        for predicate in self.predicates:
            output += predicate.print_predicate()
            if predicate is not self.predicates[-1]:
                output += ", "
        output += ")"
        if direct is not False:
            print(output)
        return output
    def deepcopy(self):
        predicates = [predicate.deepcopy() for predicate in self.predicates]
        return Clause(predicates)
```

如上，基本的数据结构就定义完了：

- KB
  - Clause1
    - Predicate1
      - name
      - arg
      - Neg
  - Clause2
    - Predicate1
      - name
      - arg
      - neg
    - Predicate2
      - name
      - arg
      - neg

下面进行字符串处理模块，字符串处理的最终目的是把字符串处理成上面类定义的对象，如下给出了两个关键函数。（为了实现这两个函数，还有一些小的函数，比较简单就不贴出来了。）

```
def parse_input(input_text):
    """input text and turns into clause object: (¬C(y), ¬L(y, rain)) """
    # Return a clause
    input_text = remove_outer_parentheses(input_text) #  ¬C(y), ¬L(y, rain)
    predicates_str = split_on_outer_comma(input_text) #  [¬C(y), ¬L(y, rain)]

    predicates = [parse_predicate(predicate_str) for predicate_str in predicates_str]
    return Clause(predicates)
```

```
def parse_predicate(predicate_str):
    """Parses a single predicate string into a Predicate object."""
    predicate_str = predicate_str.strip()
    negated = predicate_str.startswith("¬")
    if negated:
        predicate_str = predicate_str[1:]  # Remove negation symbol

    name_end_index = predicate_str.find("(")
    name = predicate_str[:name_end_index]
    arguments_str = predicate_str[name_end_index+1:-1]  # Exclude parentheses
    arguments = arguments_str.split(", ") if ", " in arguments_str else
[arguments_str]

    return Predicate(name, arguments, negated)
```

下面进入主要算法模块

首先定义了一个`visited`模块，保证每次不归结同一对谓词。

然后对KB内的clause进行循环查找。每次查找到predicate后使用`is_match`进行匹配判断。

若判断可以进行归结，则继续执行，使用`MGU`算法获取一个参数替换列表，然后使用`resolve`函数进行替换并且添加到KB中。

往复进行单步归结，然后直到`goal_test`满足或者便利一遍没有找到可以归结的子句。

```
def resolution_algorithm(KB, debug=False):
    visited = set()
    goal_test = False
    queue = deque(KB)  # Initialize the queue with the initial KB clauses
    while queue:
        clause1 = queue.popleft()
        for i, clause2 in enumerate(queue):
            for index1, predicate1 in enumerate(clause1.predicates):
                for index2, predicate2 in enumerate(clause2.predicates):
                    signal = str(i+1) + int_to_char(index1) + ", " + str(i+2) +
int_to_char(index2)
                    if is_match(predicate1, predicate2) and signal not in visited:
                        replace = MGU(predicate1, predicate2)
```

```
                        if replace != False:
                            visited.add(signal)
                            new_clause = resolve(clause1, clause2, predicate1,
predicate2, replace)
                            if debug:
                                debug_info(KB)
                            display_info(new_clause, replace, signal)
                            if len(new_clause.predicates) == 0:
                                goal_test = True
                                break
                            queue.append(new_clause)
                if goal_test:
                    break
            if goal_test:
                break
        if goal_test:
            break
    if goal_test:
        print("\nSo SATISFY\n")
    else:
        print("\nSO NOT SATISFY\n")
```

至此，所有主要算法已经介绍完毕，如下是一个测试案例

```
def test2():
    """test in dataset1"""
    input_text1 = """
11
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))
"""

    input_text2 = """
5
On(aa, bb)
On(bb, cc)
Green(aa)
¬Green(cc)
(¬On(x, y), ¬Green(x), Green(y))
"""

    input_text = input_text2.strip().split("\n")[1:]
```

```
    print("Input data: ",input_text, sep="\n")

    KB = []
    for input in input_text:
        KB.append(logic.parse_input(input))

    print("\nData loading...\n")
    print("Clause number: ", len(KB))
    for clause in KB:
        clause.print_clause()

    print("\nLoad finished, processing...\n")
    logic.resolution_algorithm(KB, debug = True)

    print("Excaution is finished, quitting...")

```

调整测试案例只需要调整输入的是`test1`还是`test2`即可。

2.4 创新点（优化）

利用正则表达式对字符串处理进行简化

```
def split_on_outer_comma(input_text):
    """Splits a string on commas that are not inside parentheses."""
    return re.split(r',(?![^()]*\))', input_text)
```

使用BFS对查找过程进行优化

```
def resolution_algorithm(KB, debug=False):
    visited = set()
    goal_test = False
    queue = deque(KB)  # Initialize the queue with the initial KB clauses
    while queue:
        clause1 = queue.popleft()
        ....
```

> 但是实际上对作用并不大，因为本身是一个非常浅的树，查找的开销也并不大。关键需要进行优化的是选择两个合适的子句，而不仅仅是更快的查找子句。

充分的Debug Info，并且提供接口决定是否显示

```python
def debug_info(KB):
    print("\n----------Debug Info------------")
    print("Number of clause in KB: ", len(KB))
    count = 1
    for clause in KB:
        print("Clause:", count,"\t",sep="",end="")
        count = count+1
        clause.print_clause()
    print("---------------------------------\n")
```

3 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

对于测试案例2

```
5
On(aa, bb)
On(bb, cc)
Green(aa)
¬Green(cc)
(¬On(x, y), ¬Green(x), Green(y))

```

运行结果如下

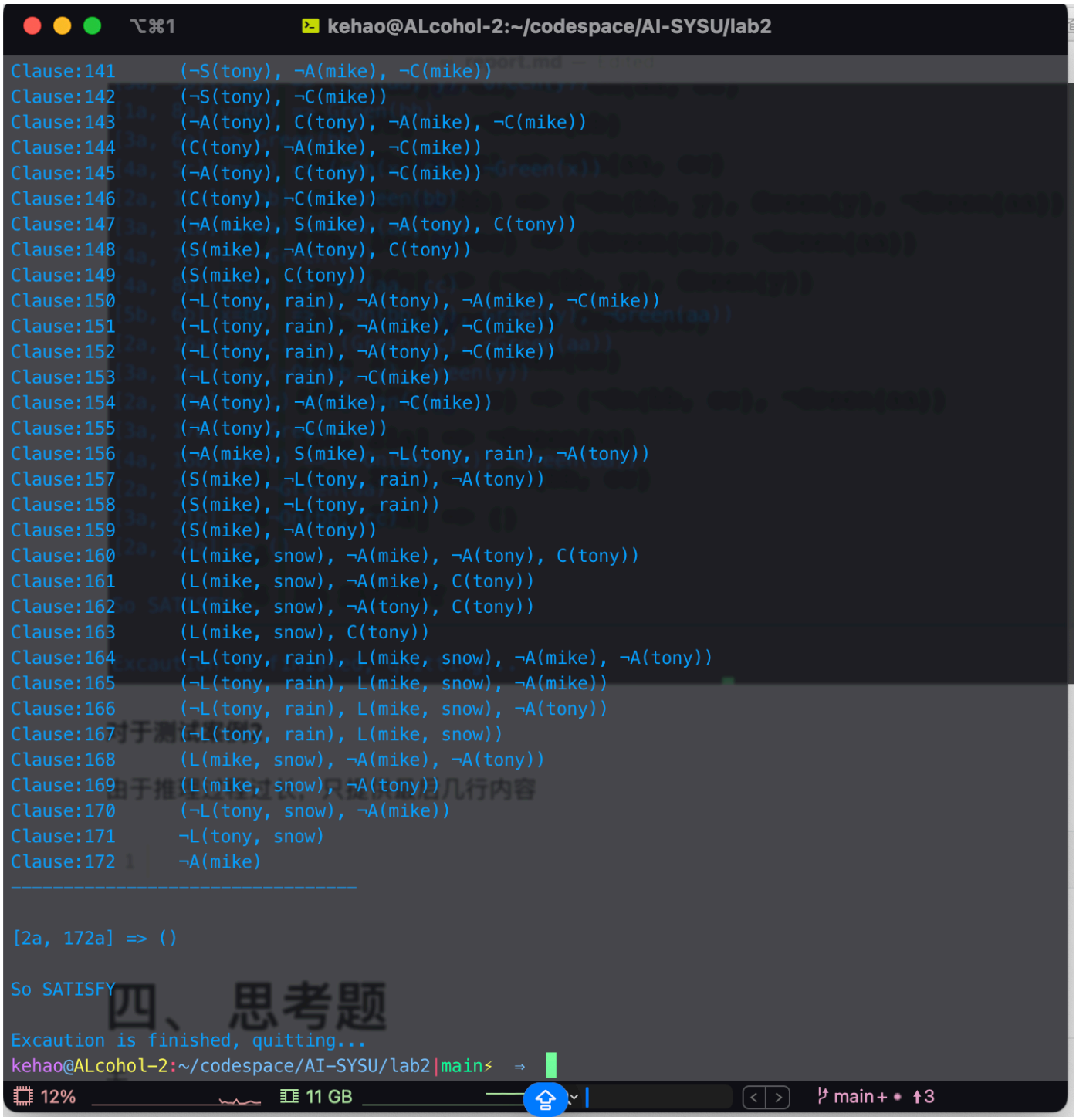
```
kehao@ALcohol-2:~/codespace/AI-SYSU/lab2|main⚡ ⇒  python test_main.py
Input data:
['On(aa, bb)', 'On(bb, cc)', 'Green(aa)', '¬Green(cc)', '(¬On(x, y), ¬Green(x), Green(y))']

Data loading...

Clause number:  5
On(aa, bb)
On(bb, cc)
Green(aa)
¬Green(cc)
(¬On(x, y), ¬Green(x), Green(y))

Load finished, processing...

[1a, 5a](x=aa)(y=bb) => (¬Green(aa), Green(bb))
[2a, 5a](x=bb)(y=cc) => (¬Green(bb), Green(cc))
[3a, 5b](x=aa) => (¬On(aa, y), Green(y))
[1a, 8a](y=bb) => Green(bb)
```

```
[3a, 6a] => Green(bb)
[4a, 5c](y=cc) => (¬On(x, cc), ¬Green(x))
[2a, 11a](x=bb) => ¬Green(bb)
[3a, 11b](x=aa) => ¬On(aa, cc)
[4a, 7b] => ¬Green(bb)
[4a, 8b](y=cc) => ¬On(aa, cc)
[5b, 6b](x=bb) => (¬On(bb, y), Green(y), ¬Green(aa))
[2a, 16a](y=cc) => (Green(cc), ¬Green(aa))
[3a, 16c] => (¬On(bb, y), Green(y))
[2a, 18a](y=cc) => Green(cc)
[3a, 17b] => Green(cc)
[4a, 16b](y=cc) => (¬On(bb, cc), ¬Green(aa))
[2a, 21a] => ¬Green(aa)
[3a, 21b] => ¬On(bb, cc)
[2a, 23a] => ()

So SATISFY
```

```
kehao@ALcohol-2:~/codespace/AI-SYSU/lab2|main⚡ ⇒ python test_main.py
Input data:
['On(aa, bb)', 'On(bb, cc)', 'Green(aa)', '¬Green(cc)', '(¬On(x, y), ¬Green(x), Green(y))']

Data loading...

Clause number:  5
On(aa, bb)
On(bb, cc)
Green(aa)
¬Green(cc)
(¬On(x, y), ¬Green(x), Green(y))

Load finished, processing...

[1a, 5a](x=aa)(y=bb) => (¬Green(aa), Green(bb))
[2a, 5a](x=bb)(y=cc) => (¬Green(bb), Green(cc))
[3a, 5b](x=aa) => (¬On(aa, y), Green(y))
[1a, 8a](y=bb) => Green(bb)
[3a, 6a] => Green(bb)
[4a, 5c](y=cc) => (¬On(x, cc), ¬Green(x))
[2a, 11a](x=bb) => ¬Green(bb)
[3a, 11b](x=aa) => ¬On(aa, cc)
[4a, 7b] => ¬Green(bb)
[4a, 8b](y=cc) => ¬On(aa, cc)
[5b, 6b](x=bb) => (¬On(bb, y), Green(y), ¬Green(aa))
[2a, 16a](y=cc) => (Green(cc), ¬Green(aa))
[3a, 16c] => (¬On(bb, y), Green(y))
[2a, 18a](y=cc) => Green(cc)
[3a, 17b] => Green(cc)
[4a, 16b](y=cc) => (¬On(bb, cc), ¬Green(aa))
[2a, 21a] => ¬Green(aa)
[3a, 21b] => ¬On(bb, cc)
[2a, 23a] => ()

So SATISFY

Excaution is finished, quitting...
```

对于测试案例2

由于推理过程过长，只提供最后几行内容



四、 思考题

无

五、 参考资料

无