中山大学计算机学院

人工智能

本科生实验报告

课程名称：Artificial Intelligence

学号	23336179	姓名	马福泉

一、 Infer.py(第三周作业一)：

1. 实验题目

1.命题逻辑的归结推理

编写函数 `ResolutionProp` 实现命题逻辑的归结推理．该函数要点如下：

- 输入为子句集(数据类型与格式详见课件)，每个子句中的元素是原子命题或其否定．
- 输出归结推理的过程，每个归结步骤存为字符串，将所有归结步骤按序存到一个列表中并返回，即返回的数据类型为 `list[str]`．
- 一个归结步骤的格式为 `步骤编号 R[用到的子句编号] = 子句`．如果一个字句包含多个公式，则每个公式用编号 `a,b,c...` 区分，如果一个字句仅包含一个公式，则不用编号区分．(见课件和例题)

例子：输入子句集

```
KB = {(FirstGrade,), (~FirstGrade,Child), (~Child,)}
```

则调用 `ResolutionProp(KB)` 后返回推理过程的列表如下：

```
1 (FirstGrade,),
2 (~FirstGrade,Child)
3 (~Child,),
4 R[1,2a] = (Child,),
5 R[3,4] = ()
```

2. 实验内容

（1） 算法原理

① 初始化

将知识库（KB）中的所有子句存储在一个列表中，保持顺序。

初始化一个集合 used_pairs，用于记录已经尝试过归结的子句对，避免重复归结。

初始化一个列表 steps，用于记录每一步的推理过程，方便输出。

② 归结推理过程：

尝试所有未归结的子句对：

如果没有新生成的子句，则考虑所有未归结的子句对。

使用两层循环生成所有可能的子句对，并检查它们是否已经在 used_pairs 中。

③ 归结操作：

对于每一对子句，调用 resolve 函数尝试归结。

resolve 函数会检查两个子句中是否存在互补文字。

如果存在，则生成一个新的子句（归结式），并返回归结式及其对应的文字索引。

如果归结式已经存在于当前的子句集中，则跳过；否则，将归结式添加到子句集中，并记录归结步骤。

④ 检查终止条件：

如果生成的归结式是空子句（()），说明找到了矛盾，知识库不可满足，推理成功，返回推理步骤。

如果没有新的子句被添加到子句集中，说明无法继续归结，推理结束。

（2） 关键代码展示

归结主循环

```
    while True:
        added = False
        pairs_to_try = []

        if newly_added_indices:
            for new_idx in newly_added_indices:
                for old_idx in range(len(new_sentences)):
                    if new_idx != old_idx and (new_idx, old_idx) not in used_pairs
and (old_idx, new_idx) not in used_pairs:
                        pairs_to_try.append((new_idx, old_idx))

        if not pairs_to_try:
            n = len(new_sentences)
            for i in range(n):
                for j in range(i+1, n):
                    if (i, j) not in used_pairs:
                        pairs_to_try.append((i, j))
```

归结逻辑

```
    resolvent, i_lit_idx, j_lit_idx = resolve(sentence_i, sentence_j)if
resolvent is not None:
        used_pairs.add((i, j))
        if resolvent not in new_sentences:
            new_sentences.append(resolvent)
            newly_added_indices.append(len(new_sentences) - 1)
```

（3） 创新点&优化

通过 `newly_added_indices` 来记录新生成的子句，并优先使用新添加的子句进行归结。具体步骤如下：

- 在归结过程中，会先检查是否有新添加的子句（存储在 `newly_added_indices` 列表中）。
- 如果有新的子句，它们会优先和已有的子句进行归结，尝试尽早使用新生成的子句进行归结，从而加快归结过程。

```
    if newly_added_indices:
```

```
        for new_idx in newly_added_indices:
            for old_idx in range(len(new_sentences)):
                if new_idx != old_idx and (new_idx, old_idx) not in used_pairs
and (old_idx, new_idx) not in used_pairs:
                    pairs_to_try.append((new_idx, old_idx))
```

效果：与理论相契合，符合我们的思考逻辑

简单高效（测试样例分析见结果）

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）

图一：

```
[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3.1.py"
1 (FirstGrade)
2 (~FirstGrade,Child)
3 (~Child)
4 R[1,2a]=(Child)
5 R[2b,3]=(~FirstGrade)
6 R[1,5]=()

[Done] exited with code=0 in 0.154 seconds

[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3.1.test.py"
1 (FirstGrade,)
2 (~FirstGrade,Child,)
3 (~Child,)
4 R[1,2a]=(Child,)
5 R[4,3]=()

[Done] exited with code=0 in 0.134 seconds
```

图二：

```
[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3.1.py"
1 (FirstGrade)
2 (~FirstGrade,Child)
3 (~Child)
4 R[1,2a]=(Child)
5 R[2b,3]=(~FirstGrade)
6 R[1,5]=()

[Done] exited with code=0 in 0.139 seconds

[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3.1.test.py"
1 (FirstGrade,)
2 (~FirstGrade,Child,)
3 (~Child,)
4 R[1,2a]=(Child,)
5 R[4,3]=()

[Done] exited with code=0 in 0.127 seconds
```

（2） 评测指标展示及分析（可分析运行时间等）

如图一图二样例所示：通过 newly_added_indices 来记录新生成的子句，并优先使用新添加的子句进行归结，对比修改之前的程序，运行速度加快。

时间复杂度：O(n^2×m)

空间复杂度：O(n^2+n×m)

其中 n 是子句的数量，m 是所有不同文字的总数。

4. 参考资料

（1）部分 debug 及优化建议参考了 ai 大模型的建议。

（2）阅读参考了以下文章：

用 python 做归结演绎推理_python 基础归结-CSDN 博客

【人工智能】鲁滨逊归结原理-Python 实现_归结原理实验-CSDN 博客

二、 MGU.py（第三周作业二）：

1. 实验题目

2.最一般合一算法

编写函数 `MGU` 实现最一般合一算法，该函数要点如下：

- 输入为两个原子公式，它们的谓词相同。其数据类型为 `str`，格式详见课件。
- 输出最一般合一的结果，数据类型为 `dict`，格式形如{变量：项，变量：项}，其中的变量和项均为字符串。
- 若不存在合一，则返回空字典。

例子：

调用 `MGU('P(xx,a)', 'P(b,yy)')` 后返回字典 `{'xx':'b', 'yy':'a'}`。

调用 `MGU('P(a,xx,f(g(yy)))', 'P(zz,f(zz),f(uu))')` 后返回字典 `{'zz':'a', 'xx':'f(a)', 'uu':'g(yy)'}`。

2. 实验内容

(1) 算法原理

① 解析原子公式

parse_atom(atom): 提取谓词和参数列表。

parse_terms(params_str): 解析参数列表，支持嵌套结构。

② 变量以及复合项判断

is_variable(term): 判断是否为变量（以小写字母开头，且非复合项）。

is_compound_term(term): 判断是否为复合项（包含括号）。

③ 变量替换

apply_replace(term, replace): 递归应用替换，将变量替换为具体值或复合项。

④ 合一算法

unify(x, y, replace):

若 x == y，直接返回替换映射 replace。

若 x 或 y 是变量，使用 unify_variable 进行合一。

若 x 和 y 是复合项，则递归合一其参数。

⑤ 变量合一逻辑

若变量已在替换映射 replace 中，则递归合一。

若变量出现在 term 中（避免循环替换），返回 None。

否则，将 var -> term 存入 replace。

⑥ MGU 计算

MGU(atom1, atom2): 计算两个原子公式的最一般合一。

若谓词不同或参数数目不同，返回 {}。

逐个参数调用 unify 进行合一，返回最终替换映射。

（2）关键代码展示（可选）

```
# 最一般合一（MGU）核心逻辑
def unify(x, y, replace):
    if replace is None:
        return None
    if x == y:
        return replace
    if is_variable(x):
        return unify_variable(x, y, replace)
    if is_variable(y):
        return unify_variable(y, x, replace)
    if is_compound_term(x) and is_compound_term(y):
        xf, xp = parse_atom(x)
        yf, yp = parse_atom(y)
        if xf != yf or len(xp) != len(yp):
            return None
        for i in range(len(xp)):
            replace = unify(xp[i], yp[i], replace)
            if replace is None:
```

```
                return None
            return replace
    return None


# 计算两个原子公式的最一般合一（MGU）
def MGU(atom1, atom2):
    pred1, params1 = parse_atom(atom1)
    pred2, params2 = parse_atom(atom2)
    if pred1 != pred2 or len(params1) != len(params2):
        return {}
    replace = {}
    for p1, p2 in zip(params1, params2):
        replace = unify(p1, p2, replace)
        if replace is None:
            return {}
    return replace
```

3. 实验结果及分析

（1）实验结果展示示例

第一个示例为实验要求，第二个示例为其他更多复杂示例。

```
[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3\lab3.2.py"
MGU('P(xx,a)', 'P(b,yy)') result is {xx=b, a=yy}
MGU('P(a,xx,f(g(yy)))', 'P(zz,f(zz),f(uu))') result is {a=zz, xx=f(zz), uu=g(yy)}

[Done] exited with code=0 in 0.135 seconds

[Running] python -u "c:\Users\86157\Desktop\AI\Code\lab3\lab3.2test1.py"
MGU('P(xx,a)', 'P(b,yy)') result is {xx=b, a=yy}
MGU('P(a,xx,f(g(yy)))', 'P(zz,f(zz),f(uu))') result is {a=zz, xx=f(zz), uu=g(yy)}
MGU('P(x,y)', 'P(a,b)') result is {x=a, y=b}
MGU('Q(x,y,z)', 'Q(a,b)') result is {}
MGU('P(x,y)', 'Q(a,b)') result is {}
MGU('P(x,f(x))', 'P(f(y),y)') result is {}
MGU('P(f(x,g(y)),h(z))', 'P(f(a,g(b)),h(c))') result is {x=a, y=b, z=c}
MGU('P(x,x)', 'P(a,b)') result is {x=a, a=b}
MGU('P(x,x)', 'P(a,a)') result is {x=a}

[Done] exited with code=0 in 0.135 seconds
```

（2） 评测指标展示及分析

① 时间复杂度分析

最优情况（变量直接匹配） $O(n)$

一般情况（递归合一） $O(n^2)$

最坏情况（深度嵌套复合项） $O(n^3)$

② 空间复杂度分析

最优情况（无递归） $O(n)$

一般情况（递归展开）	$O(n^2)$
最坏情况（深度递归 + 复杂表达式）	$O(n^2)$

4. 参考资料

（1）部分 debug，优化建议参考了 ai 大模型的建议。

（2）阅读参考了以下文章：

合一算法的 Python 实现--人工智能_unify 算法-CSDN 博客

合一算法的 Python 实现--人工智能_python 实现最一般合一-CSDN 博客

vscode 中 python 中文输出乱码问题 - Arlong - SegmentFault 思否

三、FOL.py（第四周）

1） 实验题目

作业1

输入

```
KB = {(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))}
```

输出

```
1 (A(tony),)
2 (A(mike),)
3 (A(john),)
4 (L(tony,rain),)
5 (L(tony,snow),)
6 (~A(x),S(x),C(x))
7 (~C(y),~L(y,rain))
8 (L(z,snow),~S(z))
9 (~L(tony,u),~L(mike,u))
10 (L(tony,v),L(mike,v))
11 (~A(w),~C(w),S(w))
12 R[2,11a]{w=mike} = (S(mike),~C(mike))
13 R[5,9a]{u=snow} = (~L(mike,snow),)
14 R[6c,12b]{x=mike} = (S(mike),~A(mike),S(mike))
15 R[2,14b] = (S(mike),)
16 R[8b,15]{z=mike} = (L(mike,snow),)
17 R[13,16] = []
```

作业2

输入

```
KB = {(On(tony,mike),),(On(mike,john),),(Green(tony),),(~Green(john),),(~On(xx,yy),~Green(xx),Green(yy))}
```

输出

```
1 (On(tony,mike),),
2 (On(mike,john),),
3 (Green(tony),),
4 (~Green(john),),
5 (~On(xx,yy),~Green(xx),Green(yy)),
6 R[4,5c]{yy=john} = (~On(xx,john),~Green(xx)),
7 R[3,5b]{xx=tony} = (~On(tony,yy),Green(yy)),
8 R[2,6a]{xx=mike} = (~Green(mike),),
9 R[1,7a]{yy=mike} = (Green(mike),),
10 R[8,9] = ()
```

2） 实验内容

1. 算法原理

(1) generate_identifier(literal_idx, clause_idx, clause_size)
生成子句和字面量的唯一标识符，用于在归结过程中标识特定的子句和字面量。
(2) are_complementary(lit1, lit2)
判断两个字面量是否互补，即一个字面量是否是另一个字面量的否定。
(3) resolve_clauses(clause1, clause2, idx1, idx2)
对两个子句进行归结，移除互补的字面量，生成新的子句。
(4) format_resolution_sequence(new_clause, id1, id2, substitutions)
生成归结步骤的字符串表示形式，包括使用的子句标识符和替换映射。
(5) 5substitute_clause(clause, substitutions)
对子句中的字面量应用替换映射，生成新的子句。
(6) resolution(KB)
执行归结过程，从知识库中生成新的子句，直到找到空子句或无法生成新的子句。
(7) 7update_num(num, steps, useful_steps, init_size)
(8) 更新步骤编号，确保步骤编号在简化后的步骤列表中保持一致。. extract_parents(seq)
(9) 从归结步骤字符串中提取父步骤的编号。. reassign_sequence(seq, old_num1, old_num2, new_num1, new_num2)
重新分配步骤编号，更新归结步骤字符串中的编号。
(10) simp_steps(steps, init_size)
简化归结步骤，去除冗余的步骤，只保留必要的归结步骤。
(11) solve(KB)
解决知识库中的归结问题，执行归结过程并简化步骤，返回最终的归结步骤列表

2. 关键代码展示（可选）

```
    # 归结过程
def resolution(KB):
    all_clauses = list(KB)
    support_list = [all_clauses[-1]]
    result = []
    processed_pairs = set()
    while True:
        new_clauses = []
        for i in range(len(all_clauses)):
            for j in range(i + 1, len(all_clauses)):
                if i == j:
                    continue
                clause1, clause2 = all_clauses[i], all_clauses[j]
                if (clause1, clause2) in processed_pairs:
                    continue
                if clause2 not in support_list and clause1 not in support_list:
```

```
                continue
            for lit_idx1 in range(len(clause1)):
                for lit_idx2 in range(len(clause2)):
                    lit1, lit2 = clause1[lit_idx1], clause2[lit_idx2]
                    if not are_complementary(lit1, lit2):
                        continue
                    lit1_clean = lit1.replace('~', '')
                    lit2_clean = lit2.replace('~', '')
                    mgu_dict = MGU([lit1_clean], [lit2_clean])
                    if mgu_dict is None:
                        continue
                    clause1_sub = substitute_clause(clause1, mgu_dict)
                    clause2_sub = substitute_clause(clause2, mgu_dict)
                    new_clause = resolve_clauses(clause1_sub, clause2_sub, lit_idx1, lit_idx2)
                    if new_clause in all_clauses or new_clause in new_clauses:
                        continue
                    processed_pairs.add((clause1, clause2))
                    id1 = generate_identifier(lit_idx1, i, len(clause1))
                    id2 = generate_identifier(lit_idx2, j, len(clause2))
                    seq = format_resolution_sequence(new_clause, id1, id2, mgu_dict)
                    result.append(seq)
                    new_clauses.append(new_clause)
                    if new_clause == ():
                        return result
    all_clauses.extend(new_clauses)
    support_list.extend(new_clauses)
```

3. 创新点&优化（如果有）

代码实现了**支持集策略**（Set of Support Strategy）。这种策略优先选择包含推理后句子（即支持集中的句子）的子句进行归结，从而减少了不必要的归结步骤，提高了推理效率。这种方法特别适用于处理大型知识库，因为它可以显著减少搜索空间，避免无意义的归结操作，从而更快地找到矛盾或证明知识库的一致性。详见上述函数。

3） 实验结果及分析

```
[Running] python -u "c:\Users\86157\Desktop\SYSUAILAB-main\mycode\FOL.py"
1 ('On(tony,mike)',)
2 ('On(mike,john)',)
3 ('Green(tony)',)
4 ('~Green(john)',)
5 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
6 R[4,5c]{yy=john}=('~On(xx,john)', '~Green(xx)')
7 R[3,6b]{xx=tony}=('~On(tony,john)',)
8 R[1,7]=()

[Done] exited with code=0 in 0.294 seconds

[Running] python -u "c:\Users\86157\Desktop\SYSUAILAB-main\mycode\FOL.py"
1 ('A(tony)',)
2 ('A(mike)',)
3 ('A(john)',)
4 ('L(tony,rain)',)
5 ('L(tony,snow)',)
6 ('~A(x)', 'S(x)', 'C(x)')
7 ('~C(y)', '~L(y,rain)')
8 ('L(z,snow)', '~S(z)')
9 ('~L(tony,u)', '~L(mike,u)')
10 ('L(tony,v)', 'L(mike,v)')
11 ('~A(w)', '~C(w)', 'S(w)')
12 R[8b,11c]=('L(z,snow)', '~A(w)', '~C(w)')
13 R[6c,12c]=('~A(x)', 'S(x)', 'L(z,snow)', '~A(w)')
14 R[8b,13b]=('L(z,snow)', '~A(x)', '~A(w)')
15 R[9a,14a]=('~L(mike,u)', '~A(x)', '~A(w)')
16 R[4,15a]=('~A(x)', '~A(w)')
17 R[1,16a]=('~A(w)',)
18 R[1,17]=()

[Done] exited with code=0 in 3.222 seconds
```

1. 代码在处理第二个样例时比较慢，主要是因为归结演绎推理算法的组合爆炸问题。具体来说：

子句数量多：随着知识库的增大，需要检查的子句对数量呈指数级增长。

MGU 计算复杂：每次归结时都需要计算最一般合一（MGU），这在复杂字面量下非常耗时。

重复子句检查：虽然有去重机制，但检查本身也会消耗大量时间

2. 优化思路：

启发式搜索：优先处理较短的子句，减少不必要的归结。

子句索引：快速定位互补字面量，减少比较次数。

剪枝策略：避免重复处理相同的子句对，减少计算量。

优化 MGU：减少 MGU 计算的开销。

3. 更多样例如下：

4.
```
[Running] python -u "c:\Users\86157\Desktop\AI\实验课\实验作业\23336179_马福泉_实验3pdf\code\FOL.py"
1 ('On(tony,mike)',)
2 ('On(mike,john)',)
3 ('Green(tony)',)
4 ('~Green(john)',)
5 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
6 R[4,5c]{yy=john}=('~On(xx,john)', '~Green(xx)')
7 R[3,6b]{xx=tony}=('~On(tony,john)',)
8 R[1,7]=()

[Done] exited with code=0 in 0.158 seconds

[Running] python -u "c:\Users\86157\Desktop\AI\实验课\实验作业\23336179_马福泉_实验3pdf\code\FOL.py"
1 ('F(f(a),g(b))',)
2 ('G(h(c),k(d))',)
3 ('~F(x,y)', '~G(u,v)', 'S(x,u)')
4 ('~S(w,z)', 'T(w,z)')
5 ('~T(f(a),h(c))',)
6 ('S(f(a),h(c))',)
7 R[4a,6]=('T(w,z)',)
8 R[5,7]=()

[Done] exited with code=0 in 0.19 seconds

[Running] python -u "c:\Users\86157\Desktop\AI\实验课\实验作业\23336179_马福泉_实验3pdf\code\FOL.py"
1 ('On(tony,mike)',)
2 ('On(mike,john)',)
3 ('Green(tony)',)
4 ('~Green(john)',)
5 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
6 ('~On(tony,z)', '~On(mike,z)', 'Green(z)')
7 R[4,6c]=('~On(tony,z)', '~On(mike,z)')
8 R[1,7a]=('~On(mike,z)',)
9 R[1,8]=()
```

4） 参考资料

1. **部分 debug，代码优化建议参考了 ai 大模型的建议。**

2. **阅读参考了以下文章：**

[用 Python 实现命题逻辑归结推理系统--人工智能_python 基于谓词逻辑的归结原理代码-CSDN 博客]
[基于 Python 实现的 Horn 子句归结-CSDN 博客]