同济大学

装

订

线

《人工智能课程设计》实验三

Horn 子句归结实验

姓　　名　：　刘越影

学　　号　：　2053051

专　　业　：　计算机科学与技术

完 成 日 期　：　2022 年 6 月 14 日

2022 年 6 月

1 实验概述

1.1 实验目的

熟悉和掌握归结原理的基本思想和基本方法，通过实验培养利用逻辑方法表示知识的能力，并掌握采用机器推理来进行问题求解的基本方法。

1.2 实验内容

（1）对所给问题进行知识的逻辑表示，转换为子句，对子句进行归结求解。

（2）选用一种编程语言，在逻辑框架中实现 Horn 子句的归结求解。

（3）对下列问题用逻辑推理的归结原理进行求解，要求界面显示每一步的求解过程。

破案问题：在一栋房子里发生了一件神秘的谋杀案，现在可以肯定以下几点事实：

(a)在这栋房子里仅住有A,B,C三人；

(b)是住在这栋房子里的人杀了A；

(c)谋杀者非常恨受害者；

(d)A所恨的人，C一定不恨；

(e)除了B以外，A恨所有的人；

(f)B恨所有不比A富有的人；

(g)A所恨的人，B也恨；

(h)没有一个人恨所有的人；

(i)杀人嫌疑犯一定不会比受害者富有。

为了推理需要，增加如下常识：

(j)A不等于B。

问：谋杀者是谁？

2 实验方案设计

2.1 总体思路与总体架构

本实验要求根据已知条件求解问题的答案，首先把已知条件用谓词公式表示出来，并化成相应的子句集，设该子句集的名字为 S0。把待求解的问题也用谓词公式表示出来，然后将其否定，并与一谓词 ANSWER 构成析取式。谓词 ANSWER 是一个专为求解问题而设置的谓词，其变量必须与问题公式的变量完全一致，在本实验“杀人问题”中，析取式为~Kill(u,A)∨ANSWER(u)。将该析取式并入子句集，合并后的子句集记为 S。接下来开始归结过程，将子句集 S 中的子句两两配对，判断选择的两子句是否能够互相归结。若两子句能够归结且归结出的新子句不在子句集中，则将新子句加入子句集中；若新子句正好是 ANSWER(X)的形式，其中 X 是本问题中任意常量，则代表已经归结出了结果，输出结果；若新子句不是该形式，则继续寻找可归结的子句进行归结。若子句集中不能再通过归结产生更多新的语句，说明该问题无解，输出提示即可。求解出问题答案后，可以通过回溯的方法寻找归结的关键路径，以演绎树的形式展示结果。

附加流程图描述如下：

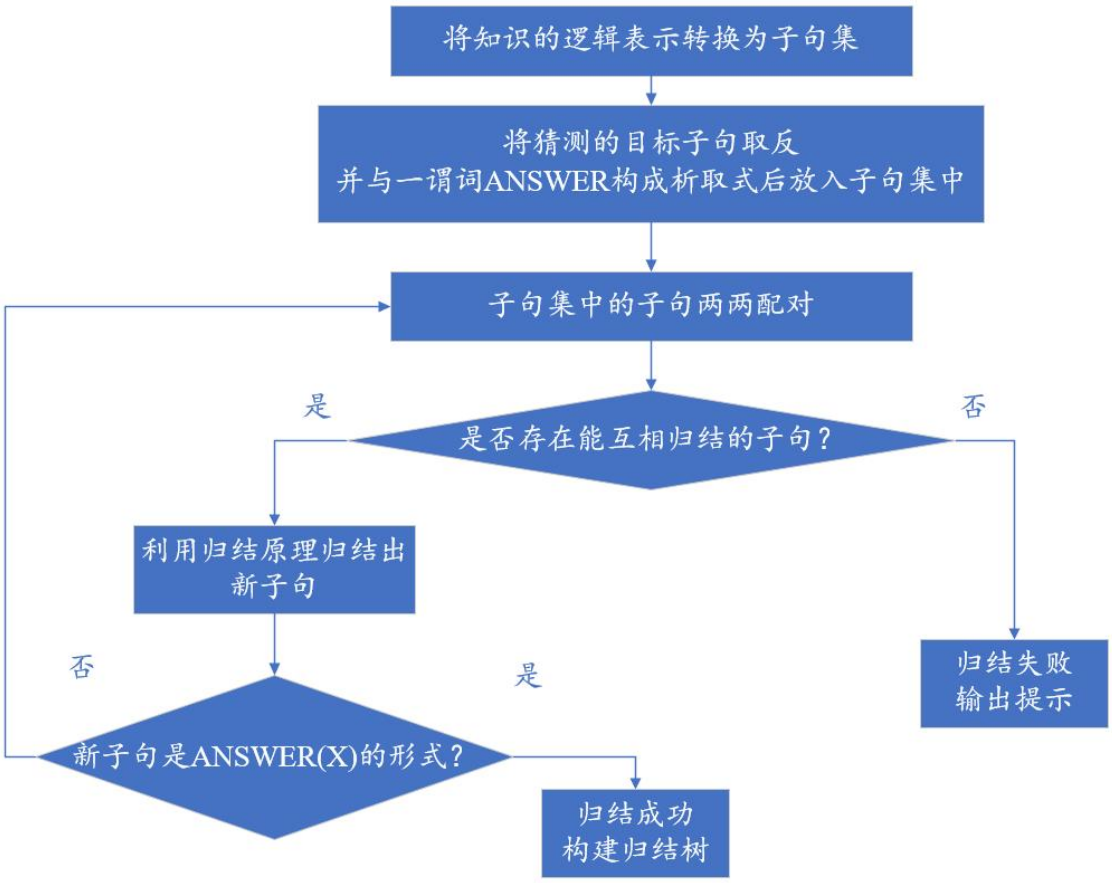

图 2.1 实验总体架构图

装 订 线

2.2 核心算法及基本原理

2.2.1 一阶逻辑化为子句

在命题逻辑中，命题之间的内在联系和数量关系并没有表达清楚，为了进一步反映这些联系，需要对命题逻辑进一步分析，分析出其中的个体词、谓词和量词，进一步研究命题之间的逻辑关系。一阶逻辑中，任何一个一阶逻辑公式都能够化成一个子句集，将初始公式化为子句集的好处是更容易利用归结原理得出目标子句。将一阶逻辑化为子句集需要遵循以下步骤：

（1）消除蕴含等价。消去蕴含符和双条件符；

（2）否定内移。将否定符号移到紧靠谓词的位置上；

（3）变量标准化。使每个量词采用不同的变量；

（4）消除存在量词。如果该存在量词不依赖于任何其它的变量，可用一个新的个体常量代替；如果存在量词是在全称量词的辖域内（如在公式$(\forall y)((\exists x)P(x, y))$中，变量 x 的取值依赖于变量 y 的取值）；

（5）将公式化为前束形。把所有全称量词移到公式的左边，并使每个量词的辖域包含这个量词后面的整个部分；

（6）化为合取范式；

（7）略去全称量词；

（8）消去合取符号；

（9）子句变量标准化。重新命名变量，使每个子句中的变量符号不同。

经过上述五个步骤，可以将初始公式化成标准的子句集形式，便于后续进行归结。

2.2.2 归结原理

归结原理又称为消解原理，由谓词公式转化为子句集的过程中可以看出，在子句集中子句之间是合取关系，其中只要有一个子句不可满足，则子句集就不可满足。若一个子句集中包含空子句，则这个子句集就是不可满足的。归结原理就是基于这一认识提出来的，通过两两子句的归结导出空子句，从而证明子句集的不可满足性。

对于命题逻辑中的任意两个子句 C_1 和 C_2。如果 C_1 中含有文字 L_1，C_2 中含有文字 L_2，并且 $L_1=\sim L_2$，那么可以从 C_1 和 C_2 中分别消去 L_1 和 L_2，并将余下的子句析取以构成一个新的子句 C_{12}，则 C_{12} 成为 C_1 和 C_2 的归结式。即 $C_{12}=(C_1-\{L_1\})\vee(C_2-\{L_2\})$，$C_1$ 和 C_2 称为 C_{12} 的亲本语句。

一阶逻辑中的归结原理与命题逻辑一致，但由于一阶逻辑将命题分为更细致的量词、谓词、变量和常量等，在一阶逻辑中应用归结原理，常常需要对个体变元进行置换，将该置换作用于给定的两个子句，使它们包含互补的文字，然后才能进行归结。举例来说，对于子句集 $S=\{P(x)\vee Q(x), \neg P(A)\vee R(y)\}$，其中的两个子句不能直接归结，但若对子句集先进行置换 $s=\{A/x\}$，则两个子句分别为 $P(A)\vee Q(A)$和 $\neg P(A)\vee R(y)$，这时再进行归结，归结结果为 $Q(A)\vee R(y)$。

利用归结反演可以实现定理的证明，在归结反演过程中，先设置目标子句，将目标子句取反放入子句集 S 中，应用归结原理对子句集中的子句进行归结，并把每次归结得到的归结式并入 S。

如此反复进行，若归结出空子句，则停止归结并证明了目标子句为真。尽管在本实验“杀人问题”中为了求出杀害 A 的凶手，我们可以依次将 Kill(A,A)，Kill(B,A)和 Kill(C,A)作为目标子句，利用归结原理进行求解，但当结论错误时，可能会由于算法始终归结不出空子句而不断运行，运行时间过长，难以得到理想的结果。所以，对于本实验的“杀人问题”，还是可以看作问题求解而不是问题证明。引入一个特殊的谓词 ANSWER，把待求解的问题用谓词公式表示出来后，将其否定与 ANSWER 构成析取式放入子句集中。如本问题要求杀害 A 的凶手，可以将该凶手记作标量 u，依照上述方法得到的析取式即为~Kill(u,A)∨ANSWER(u)。对子句集利用归结原理进行归结，在归结过程中，通过合一，改变 ANSWER 中的变元。如果得到归结式为 ANSWER(X)的形式，其中 X 是本问题中任意常量，则该问题的答案即为 X。

2.3 模块设计

2.3.1 一阶逻辑化为子句集

这一部分由依照 2.2.1 中提到的一阶逻辑化为子句集的方式人工进行处理，以本实验“杀人问题”为例，将已知前提用一阶逻辑表示并将一阶逻辑化为子句集的过程如下所示：

知识表示：令 Kill(x,y)：x 谋杀了 y；Hate(x,y)：x 恨 y；Richer(x,y)：x 比 y 富有。

将已知事实用谓词公式表示出来：

F1：Kill(A,A)∨Kill(B,A)∨Kill(C,A)

F2：Kill(x,A)→Hate(x,A) = ~Kill(x,A)∨Hate(x,A)

F3：Hate(A,x)→~Hate(C,x) = ~Hate(A,x)∨~Hate(C,x)

F4：Hate(A,A)∧Hate(A,C)

F5：∀x(~Richer(x,A)→Hate(B,x)) = Richer(x,A)∨Hate(B,x)

F6：∀x(Hate(A,x)→Hate(B,x)) = ~Hate(A,x)∨Hate(B,x)

F7：~∃x∀y Hate(x,y) =

(~Hate(A,A)∨~Hate(A,B)∨~Hate(A,C))∧(~Hate(B,A)∨~Hate(B,B)∨~Hate(B,C))∧(~Hate(C,A)∨~Hate(C,B)∨~Hate(C,C))

F8：Kill(x,A)→~Richer(x,A) = ~Kill(x,A)∨~Richer(x,A)

最终的子句集：

S={

Kill(A,A)∨Kill(B,A)∨Kill(C,A)；

~Kill(x,A)∨Hate(x,A)；

~Hate(A,x)∨~Hate(C,x)；

Hate(A,A)；

Hate(A,C)；

Richer(x,A)∨Hate(B,x)；

~Hate(A,x)∨Hate(B,x)；

~Hate(A,A)∨~Hate(A,B)∨~Hate(A,C)；

~Hate(B,A)∨~Hate(B,B)∨~Hate(B,C)；

~Hate(C,A)∨~Hate(C,B)∨~Hate(C,C)；

~Kill(x,A)∨~Richer(x,A)

}

2.3.2 输入模块

在输入模块，要求用户依次输入本问题用到的变量集合、常量集合、谓词集合、子句集以及求解目标子句，并设置了判断子句是否合法的功能，若输入的子句含有之前未设置的变量、常量或谓词，则提示重新输入，不符合格式要求的子句同样会被要求重新输入，增加了程序的鲁棒性和交互性。值得注意的是，在输入开始前，程序将特殊变量 u 和特殊谓词 ANSWER 分别放入变量集合和谓词集合中，所以程序将 u 和 ANSWER 作为保留字不允许用户使用。对于目标子句，程序自动将其取反后与 ANSWER(u)析取放入子句集中。输入模块的可视化展示如下图所示：

```
请输入所有变量标识符的集合，以#结束（u为保留字，请勿输入）：
x
#
请输入所有常量标识符的集合，以#结束：
A
B
C
#
请输入所有谓词的集合，以#结束：（ANSWER为保留字，请勿输入）
Kill
Hate
Richer
#
请输入子句集中子句数量：11
请依次输入11条子句：
（非用“!”表示；或用“|”表示）
Kill(A,A)|Kill(B,A)|Kill(C,A)
!Kill(x,A)|Hate(x,A)
!Hate(A,x)|!Hate(C,x)
Hate(A,A)
Hate(A,C)
Richer(x,A)|Hate(B,x)
!Hate(A,x)|Hate(B,x)
!Hate(A,A)|!Hate(A,B)|!Hate(A,C)
!Hate(B,A)|!Hate(B,B)|!Hate(B,C)
!Hate(C,A)|!Hate(C,B)|!Hate(C,C)
!Kill(x,A)|!Richer(x,A)
请输入此次归结的目标子句（待求变量用u表示）
Kill(u,A)
```

图 2.2 输入模块可视化展示

2.3.3 归结模块

在本实验中，为了增加归结到目标子句的速度，采用 map 结构作为子句集的数据结构，map 中的元素形式为<Clause,int>，键值对中 key 为自定义结构体子句，value 为该子句的唯一标号，map 采用自定义的排序方式，即按照子句长短进行排序。在新一轮归结中，都优先归结较短的子句，加快归结到目标子句的速度。归结过程如下图所示：

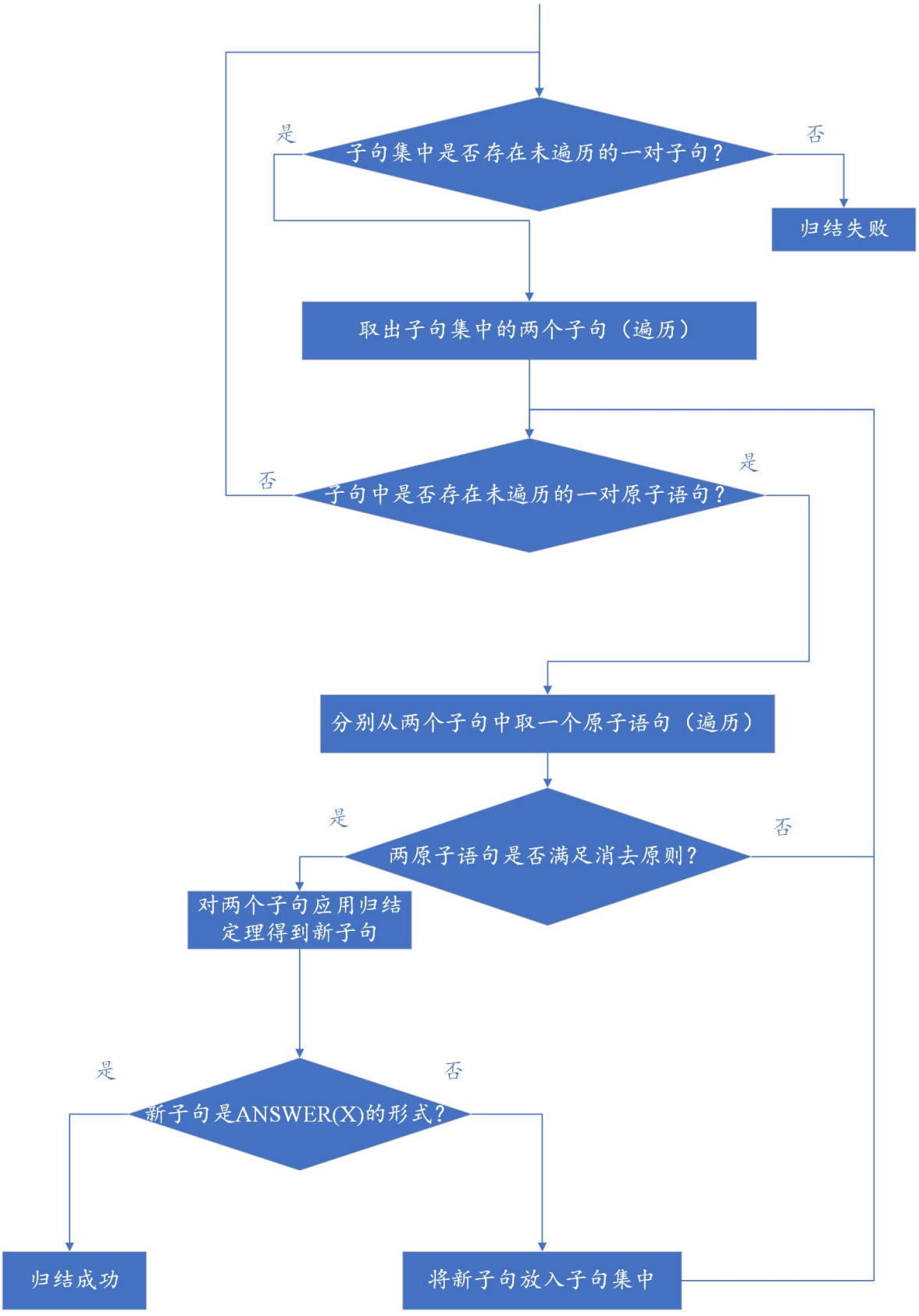

图 2.3 归结模块流程图

在归结模块的具体实现中有许多细节问题需要注意。首先是子句集的遍历，如果在不断归结产生新子句的同时遍历子句集，可能会导致位置在前面的子句无法与后面新产生的子句进行归结，进而漏掉一些可能的归结情况。同时，由于子句集采用 map 的数据结构进行存储，每次有新的子句进入子句集中时都会进行排序，子句集中原本含有的子句的位置可能会发生改变，如果采用这种遍历方式可能会造成归结顺序混乱。所以本实验采用的是**水平浸透法**，归结的步骤如下：

（1）把 S0 中的子句进行排序；

（2）在 S0 中顺序地考虑两个子句的归结式：即第一个子句与其后各个子句进行归结，然后第二个子句与其后各个子句进行归结，第三个子句与其后各个子句进行归结...；直至倒数第二个子句与最后一个子句进行归结，得到子句集 S1，S1={C12|C1∈S0，C2∈S0}，检查 S1 中是否有目标形式的子句，若有则归结结束，否则继续下一步骤；

（3）将 S1 并入 S0 得到 S0∪S1，再顺序地考虑子句集 S0∪S1 与 S1 的归结式，即一个子句来自 S0∪S1，另一个子句来自 S1，得到子句集 S2，S2={C12|C1∈S0∪S1，C2∈S1}，检查 S2 中是否有目标形式的子句，若有则归结结束，否则继续上述过程.....

采用水平浸透法既能够避免归结过程中大量无用子句，又能够防止遗漏某些子句结果。

部分归结过程如下图所示，每一条归结展示了新产生的子句的两个亲本子句：

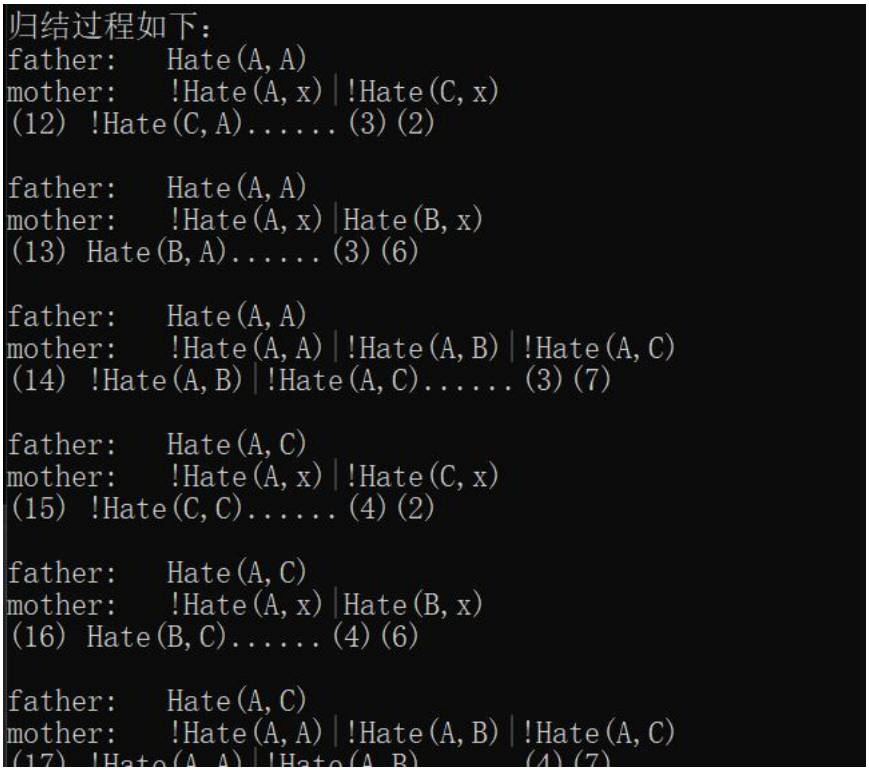

图 2.4 部分归结过程可视化展示(1)

```
father:   Kill(A,A)|Kill(B,A)|!Hate(B,A)|!Hate(B,B)
mother:   Kill(A,A)|!Hate(C,B)|!Hate(C,C)|Hate(B,A)
(2194) Kill(A,A)|Kill(B,A)|!Hate(B,B)|!Hate(C,B)|!Hate(C,C)......(176)(172)

father:   Kill(A,A)|Kill(B,A)|!Hate(B,A)|!Hate(B,B)
mother:   Kill(C,A)|!Hate(A,B)|!Hate(A,C)|Hate(B,A)
(2195) Kill(A,A)|Kill(B,A)|Kill(C,A)|!Hate(A,B)|!Hate(A,C)|!Hate(B,B)......(176)(164)

father:   Kill(A,A)|Kill(C,A)|!Hate(B,A)|!Hate(B,C)
mother:   Kill(A,A)|!Hate(C,B)|!Hate(C,C)|Hate(B,A)
(2196) Kill(A,A)|Kill(C,A)|!Hate(B,C)|!Hate(C,B)|!Hate(C,C)......(181)(172)

father:   Kill(A,A)|Kill(C,A)|!Hate(B,A)|!Hate(B,C)
mother:   Kill(C,A)|!Hate(A,B)|!Hate(A,C)|Hate(B,A)
(2197) Kill(A,A)|Kill(C,A)|!Hate(A,B)|!Hate(A,C)|!Hate(B,C)......(181)(164)

father:   !Kill(B,A)
mother:   ANSWER(A)|Kill(B,A)
(2198) ANSWER(A)......(247)(184)
归结成功!
```

图 2.5 部分归结过程可视化展示(2)

2.3.4 回溯模块

归结模块产生了许多与结果无关的子句，为了更清晰地展示归结出结果的过程，可以从目标子句 ANSWER(X)开始回溯，找子句的两个亲本，再找亲本的两个亲本，利用广度优先搜索的思想可以找出归结路径上的关键结点，进而得到关键路径如下图所示（其中 father_id 或 mother_id 等于-1 代表该子句是原始子句集中无亲本的子句）：

```
关键路径如下:
father_id: (-1)  mother_id: (-1)
(2) !Hate(A,x)|!Hate(C,x)
father_id: (-1)  mother_id: (-1)
(3) Hate(A,A)
father_id: (-1)  mother_id: (-1)
(8) !Hate(B,A)|!Hate(B,B)|!Hate(B,C)
father_id: (-1)  mother_id: (-1)
(5) Hate(B,x)|Richer(x,A)
father_id: (-1)  mother_id: (-1)
(10) !Kill(x,A)|!Richer(x,A)
father_id: (-1)  mother_id: (-1)
(0) Kill(A,A)|Kill(B,A)|Kill(C,A)
father_id: (-1)  mother_id: (-1)
(11) !Kill(u,A)|ANSWER(u)
father_id: (-1)  mother_id: (-1)
(1) !Kill(x,A)|Hate(x,A)
father_id: (3)  mother_id: (2)
(12) !Hate(C,A)
```

```
father_id: (1)  mother_id: (8)
(28) !Kill(B,A)|!Hate(B,B)|!Hate(B,C)
father_id: (10)  mother_id: (5)
(30) !Kill(x,A)|Hate(B,x)
father_id: (-1)  mother_id: (-1)
(6) !Hate(A,x)|Hate(B,x)
father_id: (-1)  mother_id: (-1)
(4) Hate(A,C)
father_id: (11)  mother_id: (0)
(18) ANSWER(A)|Kill(B,A)|Kill(C,A)
father_id: (12)  mother_id: (1)
(40) !Kill(C,A)
father_id: (30)  mother_id: (28)
(102) !Kill(B,A)|!Kill(B,A)|!Hate(B,C)
father_id: (4)  mother_id: (6)
(16) Hate(B,C)
father_id: (40)  mother_id: (18)
(184) ANSWER(A)|Kill(B,A)
father_id: (16)  mother_id: (102)
(247) !Kill(B,A)
father_id: (247)  mother_id: (184)
(2198) ANSWER(A)
```

图 2.6 关键路径展示

为了可视化展示归结过程，将上述回溯的内容构建成演绎树的形式。利用 dot 语言输出重定向至 tree.dot 文件中。最后通过 tree.bat 文件运行 tree.dot 文件，生成可视化的 tree.png, 便于用户查看。tree.dot 文件部分内容以及 tree.png 中搜索树可视化展示如下：

tree.dot - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
digraph {
node[shape = box]
2[label="!Hate(A,x)|!Hate(C,x)"];
3[label="Hate(A,A)"];
8[label="!Hate(B,A)|!Hate(B,B)|!Hate(B,C)"];
5[label="Hate(B,x)|Richer(x,A)"];
10[label="!Kill(x,A)|!Richer(x,A)"];
0[label="Kill(A,A)|Kill(B,A)|Kill(C,A)"];
11[label="!Kill(u,A)|ANSWER(u)"];
1[label="!Kill(x,A)|Hate(x,A)"];
12[label="!Hate(C,A)"];
28[label="!Kill(B,A)|!Hate(B,B)|!Hate(B,C)"];
30[label="!Kill(x,A)|Hate(B,x)"];
6[label="!Hate(A,x)|Hate(B,x)"];
4[label="Hate(A,C)"];
18[label="ANSWER(A)|Kill(B,A)|Kill(C,A)"];
40[label="!Kill(C,A)"];
102[label="!Kill(B,A)|!Kill(B,A)|!Hate(B,C)"];
16[label="Hate(B,C)"];
184[label="ANSWER(A)|Kill(B,A)"];
247[label="!Kill(B,A)"];
2198[label="ANSWER(A)"];
247->2198;
184->2198;
16->247;
102->247;
40->184;
18->184;
4->16;
6->16;
30->102;
28->102;
12->40;
1->40;
11->18;
0->18;
10->30;
5->30;
1->28;
8->28;
3->12;
2->12;
}
```

图 2.7 tree.dot 文件内容展示

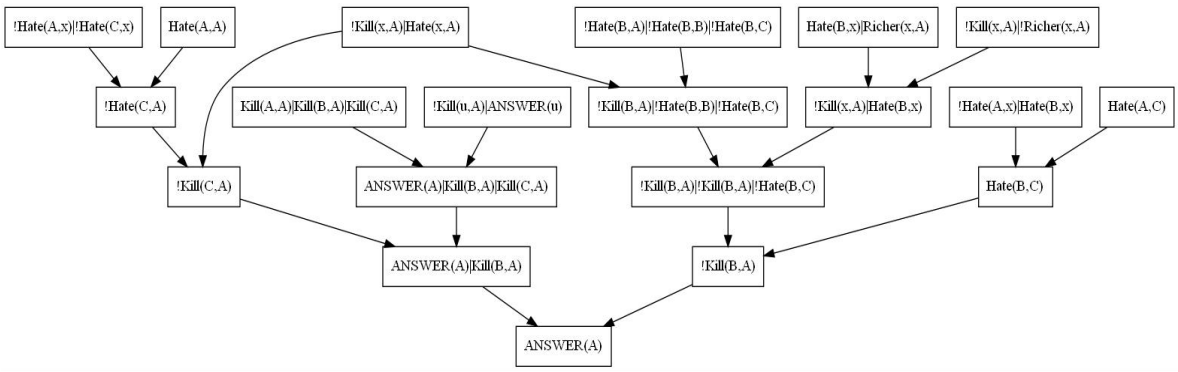

图 2.8 tree.png 演绎树展示

2.4 其它创新内容或优化算法

归结原理在定理证明、答案求解方面确实很有用处，但在实际实现中却会遇到很多问题，最典型的问题就是在归结过程中容易产生大量重复的无用子句。本实验采用水平浸透法，结合 map 数据结构本身所具有的排序功能，尽可能以最快的速度接近目标。同时，在新子句进入子句集之前，也进行判重处理，防止生成重复的语句。

3 实验过程

3.1 环境说明

本实验的环境说明如下：

表 3.1 实验环境说明

操作系统	Windows 10
开发语言	C++
开发环境及具体版本	Microsoft Visual Studio Community 2019 版本 16.11.6
核心使用库	<iostream> <vector> <iterator> <map> <algorithm> <fstream> <algorithm>

3.2 源代码文件清单、主要结构体和主要函数清单

3.2.1 源代码文件清单

表 3.2 源代码文件清单

文件名称	描述或功能
horn.h	结构体的声明；全局函数的声明
horn.cpp	结构体中函数的实现；全局函数的实现
main.cpp	主函数

3.2.2 identifier 结构体

由于原子语句中可能有常量和变量，为了将常量和变量统一处理，建立 identifier 结构体，其数据字段如下表所示：

表 3.3 identifier 结构体数据字段

数据字段	描述
bool category	区分该标识符是常量（CONSTANT） 还是变量（ VARIABLE）
int id	常量或变量对应集合的下标

3.2.3 Atom 结构体

该结构体用来说明原子语句的属性，包括以下数据字段：

表 3.4 Atom 结构体数据字段

数据字段	描述
bool positive	带否定符号则为 0；不带否定符号则为 1
int predicate_id	谓词对应索引
vector<identifier> element	按顺序列出原子式中的标识符

同时，还包括以下方法：

（1）bool erase_leagal(const Atom& a, vector<int>&change_list)

函数功能：判断两个原子语句能否互相消解。若不能返回 false，若能返回 true，且 change_list 表里记录着合一置换对应标识符下标。

参数说明：a：参加判断是否能够消解的原子语句

change_list：返回的合一置换表

3.2.4 Clause 结构体

该结构体用于说明子句所具有的相关属性，包括以下数据字段：

表 3.5 Clause 结构体数据字段

数据字段	描述
int source_1	第一个亲本子句对应 id
int source_2	第二个亲本子句对应 id
vector<Atom> atom_set	按顺序列出子句中的原子式

3.2.5 部分全局函数清单

（1）int find_predicate(string s)

函数功能：在谓词表中查找谓词，若找到返回索引，否则返回-1。

参数说明：s：待查找的谓词对应字符串

（2）int find_variable(string s)

函数功能：在变量表中查找变量，若找到返回索引，否则返回-1。

参数说明：s：待查找的变量对应字符串

（3）int find_constant(string s)

函数功能：在常量表中查找常量，若找到返回索引，否则返回-1。

参数说明：s：待查找的谓词对应字符串

（4）int atom_exist(Atom atom, vector<Atom> atom_set)

函数功能：在原子语句集合中查找原子语句，若找到返回索引，否则返回-1。

参数说明：atom：待查找的原子语句

atom_set：待查找的原子语句集合

（5）int clause_exist(Clause clause)

函数功能：在子句集中查找子句，若找到返回索引，否则返回-1。

参数说明：clause：待查找的子句

（6）bool atom_legal(string s, Atom& ret)

函数功能：判断某字符串是否是合法的原子语句。若不合法，返回 false；若合法，返回 true，同时将字符串转换成自定义结构体 Atom 的形式通过 ret 返回。

参数说明：s：待判断的字符串

ret：将合法字符串转换成的 Atom 结构体

（7）bool clause_legal(string s, Clause& ret)

函数功能：判断某字符串是否是合法的子句。若不合法，返回 false；若合法，返回 true，同时将字符串转换成自定义结构体 Clause 的形式通过 ret 返回。

参数说明：s：待判断的字符串

ret：将合法字符串转换成的 Clause 结构体

（8）void read_in()

函数功能：读入变量、常量、谓词、子句集。

（9）bool comp_atom(const Atom& a1,const Atom& a2)

函数功能：定义两个原子语句的比较运算。

参数说明：a1：参与比较的第一个原子语句结构体

a2：参与比较的第二个原子语句结构体

（10）bool judge_target(Clause clause)

函数功能：判断某子句是否为想要的结果。若是则返回 true，否则返回 false。

参数说明：s：待判断的子句结构体

（11）bool guijie_clause(int father_id, int mother_id)

函数功能：对两个子句进行归结。若归结出目标子句返回 true，否则返回 false。

参数说明：father_id：第一个亲本子句在子句集中对应 ID

mother_id：第二个亲本子句在子句集中对应 ID

（12）bool guijie()

函数功能：对子句集中的子句进行归结（若成功返回 1；若失败返回 0）。

（13）void trace_back()

函数功能：对子句集 clause_set 中的子句进行回溯，找出关键路径上的子句，并利用重定向输出 dot 语言至文件中生成演绎树。

此外，为更方便地使用自定义结构体 identifier，Atom 和 Clause，程序中还重载了一系列运算符以及输入输出符，在此不进行赘述。

装 订 线

4 总结

4.1 实验中存在的问题及解决方法

（1）本实验最开始实现的逻辑是利用归结反演的规则，将目标子句取反放入子句集中，在子句集中统一，目标是归结出空子句，通过矛盾的产生反过来说明结论的正确性。这种方法对于较为简单的例子很快能够实现，无论是否能够归结出结果，均能很快得出结果。但对于本实验“杀人问题”这种子句集较为庞大且子句较长的测试用例来说，却很难在期望时间内归结出答案，尤其是结论错误时，得出“归结失败”结论需要程序运行很长时间。解决方法是将本题当作求解问题而不是证明判断问题，引入一个特殊的谓词 ANSWER，把待求解的问题用谓词公式表示出来后，将其否定与 ANSWER 构成析取式放入子句集中，目标子句是 ANSWER(X)形式的子句。既能够直接得出答案，又能提高程序运行的效率。

（2）为了进一步提高加快归结出目标子句的速度，思考可以在每一轮归结前对子句集中的子句按长度进行排序，将长度较短的子句放在前面优先处理，因为目标子句中只有一个原子语句，从较短子句中归结出目标子句的概率显然比从较长子句中归结出目标子句的概率更大。本实验最开始采用 vector 结构对子句集进行存储，要实现排序需要自定义子句结构体 Clause 的排序函数，但这样会导致一个问题，就是每次排序子句对应的索引都会发生变化。在最后回溯关键路径时不能通过 Clause 结构体中记录的 father_id 和 mother_id 实现。解决方法是改用 map 结构存储子句集，作为一种关联容器，map 提供一对一的 hash，能够以键值对的形式存储数据。同时，map 内部自建红黑树，对数据有自动排序功能。在本实验中，可以定义键值对<Clause,int>，以子句结构体作为 key，以子句唯一 ID 作为 value，自定义 Clause 排序函数实现，成功解决问题。

4.2 心得体会

通过此次 horn 子句归结实验，我熟悉和掌握了归结原理的基本思想和基本方法，培养了利用逻辑方法表示知识的能力，并掌握了采用机器推理来进行问题求解的基本方法。针对“杀人问题”，得出了凶手是 A 的结论。本实验的涉及到的算法原理并不难，但在实现时却遇到了许多问题，前后历时三天才完成。主要原因是对 C++ STL 中各种容器的相关操作不够熟悉，如自定义 sort 函数必须满足严格弱序，需要利用迭代器对 map 进行遍历，map 如何按照 key 进行排序等等。这些问题造成的 bug 在程序中比较隐蔽，只能通过逐步调试，深入检查每一个函数的每一条语句才能发现。由于程序还涉及很多自定义结构体的运算符重载，我对多态的认识也进一步加深。

4.3 后续改进方向

（1）查找文献发现，要进一步提高归结效率，可以采用一系列归结控制策略，如纯文字删除策略、重言式删除策略、包孕删除策略、线性归结策略等，后续可以进一步改进；

（2）由于时间原因，本程序基于控制台实现，后续可以尝试设计更美观的可视化界面；

4.4 成员分工与自评

本次实验全部由刘越影同学独立完成。本实验预期的功能均成功实现，实验完成度高，实验结果质量较好。但本实验还有很多拓展方向，后续可以继续探索改进。

参考文献

[1] (美) 罗素, (美) 诺维格. 人工智能：一种现代的方法[M]. 清华大学出版社, 2013.

[2] 肖启莉, 肖启敏. 归结原理及其应用[J]. 计算机与数字工程, 2007(05):183+187+212.

[3] 刘素姣. 一阶谓词逻辑在人工智能中的应用[D]. 河南大学, 2004.

[4] 杜辉. 基于归结原理的程序综合设计与实现[D]. 大连理工大学, 2006.

[5] 宁欣然. 命题逻辑和一阶逻辑子句集的冗余性研究[D]. 西南交通大学, 2019.