

一. 实验概述

(1) 实验目的

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

(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？
4. 撰写实验报告，提交源代码（**进行注释**）、实验报告、汇报 PPT

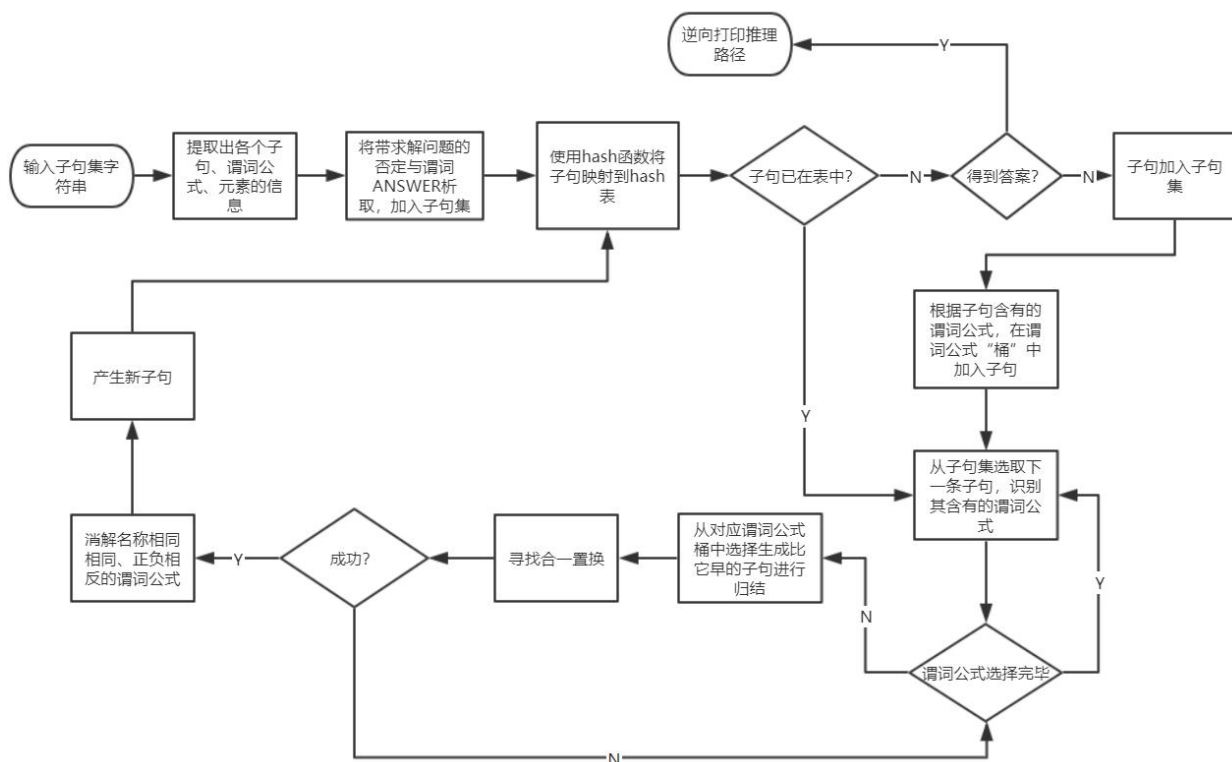
二. 实验方案设计

(1) 总体设计思路与总体架构

总体设计框架分为三大部分：

- ① 信息处理部分：将读入的字符串分解，转化为子句-谓词-元素的三层结构存储在对应类中，从而得到初始子句集。
- ② 归结部分：使用前向连接，根据归结原理对子句集不断进行归结，直至得到含有单一的 **ANSWER** 谓词子句位置，求解答案完成。
- ③ 打印部分：得到了单一的 **ANSWER** 谓词(即求解答案完成)时从最后一条生成的子句逆向寻找归结路径，打印出有效的归结过程，从而得到答案的每一步求解过程。

流程图如下：



(2) 核心算法及基本原理：

归结原理求答案：

- ① 把已知前提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 **S**。
- ② 把带求解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。
- ③ 把②中的析取式化为子句集，并把该子句集与 **S** 合并成子句集 **T**。
- ④ 对子句集中的两条子句，若其含有互补的文字且互补文字能够进行合一置换，则将两条子句析取，消除对应互补文字后得到的子句加入子句集中。通过合一可以改变 **ANSWER** 中的变元

- ⑤ 当得到仅含有一个谓词公式 **ANSWER** 的子句时，代表答案求解完成，问题的答案即在 **ANSWER** 谓词中。归结过程结束。

合一置换：

两个表达式匹配当且仅当其语法是等价的，即谓词公式的名称、谓词公式的元素(项)种类、名称和数量对应相同。谓词公式的元素为常量、变量和函数。合一通过寻找元素对变量的置换而使表达式一致。置换用有序对的集合 $S=\{t_1/v_1, t_2/v_2, \dots, t_n/v_n\}$ 来表示，其中 t_i/v_i 表示将表达式中所有的变量 v_i 用元素(项) t_i 代替， t_i 可以是变量、常量或函数。特别地，一个变量不能用含有同一变量的元素(项)来代替。

(3) 模块设计：本实验的具体模块设计

①信息处理与存储模块：

对输入的每一个字符串，以析取符号为分隔，每一小段代表了一个谓词公式。对于每一个谓词公式，在第一个 '(' 出现之前的部分是谓词公式的名称。与第一个出现的 '(' 对应的 ')' 之前、第一个出现的 '(' 之后的部分，以逗号为分隔，每一小段代表了一个元素(项)，对于函数元素，其又包含内层括号，括号里为函数的变量。

通过上述规则对读入的字符串进行处理，可以提取出每一个子句、谓词公式、元素(项)的信息。通过 **horn** 类(含有 **point** 类成员)、**point** 类(含有 **vari** 数组成员)、**vari** 数组分别存储子句、谓词公式、元素(项)，实现对知识库的获取和表示。

②hash 模块：

对每一个成功提取出的子句信息，为了快速判断后续是否有与之完全相同的子句重复出现，需要使用 **hash** 将子句映射到 **hash** 表中，并解决冲突(详细的映射和冲突解决过程参见后面)，若某次冲突时，在冲突位置的子句和正在 **hash** 的子句结构内容完全相同(通过重载 **horn** 类的 **==** 符号实现)，则 **hash** 失败，该子句已经出现过，不再加入子句集中。

③归结模块：

每次从子句集中按顺序选取一个尚未归结的子句尝试与子句集中其他的子句消去互补文字，归结形成新的子句。根据选取的子句所含有的谓词公式，到对应的谓词公式“桶”中存储的子句中寻找归结对象(谓词公式“桶”中的子句均含有对应谓词公式)，并只和在该子句之前生成的子句尝试归结以避免对同一对子句重复尝试归结。若新形成的子句尚未出现过，则加入子句集。直至出现单一 **ANSWER** 谓词公式的子句。

④合一置换模块：

对于将要归结的含有互补文字的一对子句，寻找一组置换使得它们匹配(合一)。并扫描互补文字的元素(项)，对于不相同的元素(项)，若有一方为变量，则将变量置换

为另一个子句对应位置的元素，并将该置换应用至进行归结的一对子句的整个子句(所有谓词公式)。特别地，需要进行发生检验，即若此时的置换是用含有相同变量的项对变量进行置换，则该置换无法进行。若最终这对子句的互补文字能够完成匹配，则合一置换成功。

⑤打印模块

得到单一的 ANSWER 谓词公式的子句后，需要从产生的众多归结过程中寻找到最终产生答案的一条推理路径。通过逆向递归过程，从最后一条归结开始(即得到答案的那一次归结)，根据其是由哪两条子句归结得到，依次递归处理这两条子句，特别地，若是由最初提供的子句归结得到则直接输出即可。

(4) 其他创新内容或优化算法

①hash 优化：

为了判断一条新得到的子句是否已经在子句集中，一种简单的方法是依次遍历子句集中的全部子句，判断是否存在重复。然而由于前向连接的无目的性，会产生大量的冗余子句，使得子句集过于庞大，单纯遍历子句集会使得效率极大降低，难以接受。因此采用 hash 的方式，对子句使用 hash 函数将其映射到 hash 表中。具体的 hash 方式如下：

首先规定字符串的 cal 值：字符串每个字符的 ASCII 值乘以对应位置编号，累和并对表长取模。其次，对子句的每一个谓词公式，将其字符串的 cal 值乘以谓词公式编号(若是负谓词则再乘以-1)，累和后取绝对值，并对 hash 表长取模。对每一个谓词公式的元素(项)，将其字符串的 cal 值乘以元素编号，累和并加上之前的结果，对表长取模。如果有函数元素，则对每一个函数的变量，将其字符串的 cal 值乘以变量编号，累和并加上之前的结果，对表长取模。最终得到的结果即为子句的 hash 函数值。当得到的 hash 函数值的位置已经存在子句时，若存在的子句与该子句相同则证明存在重复，否则使用线性探测，逐渐后移 hash 位置并重复检查过程，直至找到空位置。

②谓词公式“桶”优化：

选取一条子句后，需要寻找到能与之归结的子句。如果遍历一遍子句集以寻找能够归结的子句，则同样极大降低了效率。可以通过保证只对那些有合一成功机会的语句进行尝试归结。对每一种谓词公式(正、负视为两种)建立一个存储桶，桶中存放了所有含有该谓词公式的子句的编号。每次归结时只需要从当前子句含有的谓词公式的互补文字对应的存储桶中的子句选取即可，桶外的子句由于不含有互补文字，完全没有归结可能，可以直接忽略。

③子句长度限制

该方法的灵感来源于深度受限搜索。由于前向连接的归结过程没有目的性，产生大量冗余子句，盲目地进行归结。可以预见，这些盲目归结有很大可能使得子句越来越长，而实际上有效的归结，子句长度(含有的谓词公式数量)总体应该变短。因此，设置子句

的限制长度，对于超出该长度的子句视为冗余而不进行归结。从而实现了对大量无效归结的剪枝。如果无法得出问题求解的结果，则可以适当增加限制长度并再次归结。

三. 实验过程

(1) 环境说明：

本实验使用 Windows 10 操作系统，使用 C++语言在 Visual Studio 2019 (v142)平台上开发，核心使用库是 `iostream`、`string`、`ctime` 等。

(2) 源代码文件清单，主要函数清单

`guijie.cpp`: 程序源文件，实现了上述所有模块和算法。

`完整过程.txt`: 输出结果文件，展现了实验要求求解的问题的全部归结过程以及有效推理路径。

`int operator==(const point& p1, const point& p2)`，重载的 `point` 类相等的判断，用于辅助判断子句是否完全相同。

`int operator==(horn t1, horn t2)`，重载的 `horn` 类相等的判断，用于判断两个子句是否完全一样。

`ostream& operator<<(ostream& out, const point& p)`，重载<<，方便输出一个谓词公式。

`ostream& operator<<(ostream& out, const horn& h)`，重载<<，方便输出一个子句。

`void deal(string st, horn& h)`，从输入的字符串(`horn` 子句)中，做语法分析，从中获得每条子句的谓词公式、公式内的元素的信息(个数、类型、名称等)。

`void add(const horn& h, int x)`，对新的子句，根据其所含有的谓词公式的正负性以及名称，将其加入对应的(多个)谓词公式"桶"。

`int cal(const string& st)`，hash 函数的一部分，对于一个字符串求得对应函数值。

`int HASH(const horn& h)`，hash 函数，用于将子句通过 hash 函数映射到 hash 表，hash 函数根据子句的各个字符串计算函数乘以字符串位置加权之和来计算。

`void change(horn& t, string s1, string s2, const point& tt, int x)`，对整个子句应用寻找到的合一置换。

`int merge(int p1, int s1, int p2, int s2)`，归结编号为 `p1` 和 `s1` 的子句，消解 `p1` 的编号为 `p2` 的谓词公式和 `s1` 的编号为 `s2` 的谓词公式。

`void result(int x)`，逆向打印出解，根据编号为 `x` 的子句是由哪两个编号的子句归结而成的，不停向上溯源，如果是由输入的条件归结而成的就直接输出。

(3) 实验结果展示

1.先使用较为简单的问题来验证程序的正确性:

(a)任何兄弟都有同一个父亲

(b)John 和 Peter 是兄弟, 且 John 的父亲是 WT

问: Peter 的父亲是谁?

根据问题可以得出如下子句集:

Brother(John,Peter)

Brother(Peter,John)

Father(Peter,WT)

$\sim\text{Brother}(x1,x2) \mid \sim\text{Father}(x1,x3) \mid \text{Father}(x2,x3)$

要求解的问题为:

Father(John,u)

依次输入初始子句的数量和内容, 并输入要求解的问题。完毕之后程序将自动进行归结, 进行问题求解。

求解得到的结果如下所示:

```
Microsoft Visual Studio 调试控制台
请输入要归结的子句个数:
4
请输入要归结的4个子句<各个变量用xm表示>
Brother(John,Peter)
Brother(Peter,John)
Father(Peter,WT)
~Brother(x1,x2) | ~Father(x1,x3) | Father(x2,x3)
请输入要求解的问题<u代表假设变量>:
Father(John,u)
归结过程如下:
第1步:
<1>Brother(Peter,John)和<3>Father(x2,x3) | ~Brother(x1,x2) | ~Father(x1,x3)归结得到<6>~Father(Peter,x3) | Father(John,x3)
第2步:
<2>Father(Peter,WT)和<6>~Father(Peter,x3) | Father(John,x3)归结得到<11>Father(John,WT)
第3步:
<4>ANSWER(u) | ~Father(John,u)和<11>Father(John,WT)归结得到<35>ANSWER(WT)
归结成功!
共计时: 0s
C:\Users\ DELL\source\repos\horn子句归结原理实验\Debug\horn子句归结原理实验.exe (进程 10776)已退出, 返回代码为: 0。
按任意键关闭此窗口...
```

最终得到 Peter 的父亲是 WT。

2.之后进行本实验要求的问题求解:

首先根据实验问题的描述, 得出如下初始子句集:

Hate(A,A)

Hate(A,C)

$\sim \text{Hate}(A, B)$
 $\sim \text{Hate}(A, x4) \mid \text{Hate}(B, x4)$
 $\sim \text{Kill}(x1, A) \mid \text{Hate}(x1, A)$
 $\sim \text{Hate}(A, x2) \mid \sim \text{Hate}(C, x2)$
 $\sim \text{Kill}(x6, A) \mid \sim \text{Rich}(x6, A)$
 $\text{Rich}(x3, A) \mid \text{Hate}(B, x3)$
 $\text{Kill}(A, A) \mid \text{Kill}(B, A) \mid \text{Kill}(C, A)$
 $\sim \text{Hate}(x5, A) \mid \sim \text{Hate}(x5, B) \mid \sim \text{Hate}(x5, C)$

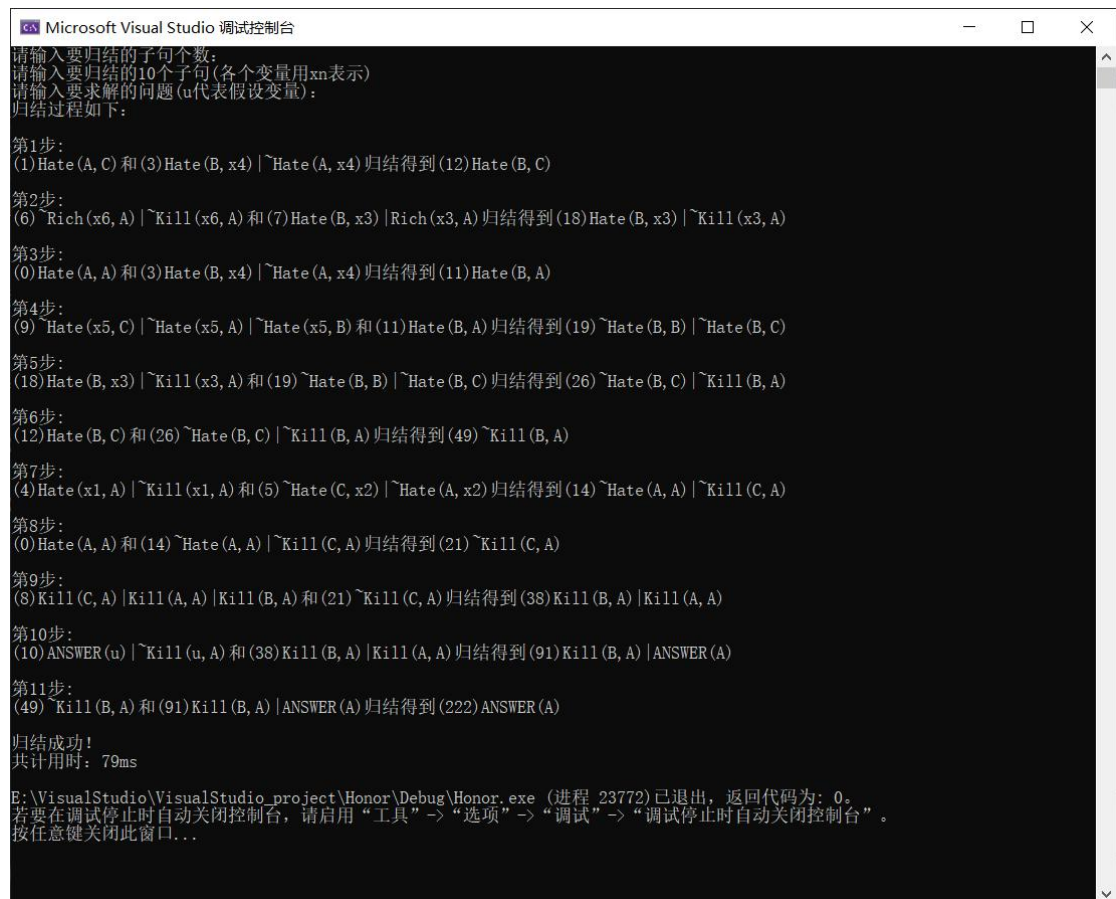
其中 $\text{Hate}(a, b)$ 代表 a 恨 b, $\text{Kill}(a, b)$ 代表 a 杀了 b, $\text{Rich}(a, b)$ 代表 a 比 b 富有。‘ \mid ’ 表示析取, ‘ \sim ’ 代表非。

要求解的问题为:

$\text{Kill}(u, A)$

依次输入初始子句的数量和内容, 并输入要求解的问题。完毕之后程序将自动进行归结, 进行问题求解。

求解得到的结果如下所示:



```
Microsoft Visual Studio 调试控制台
请输入要归结的子句个数:
请输入要归结的10个子句(各个变量用xn表示):
请输入要求解的问题(u代表假设变量):
归结过程如下:

第1步:
(1) Hate(A, C) 和 (3) Hate(B, x4) | ~Hate(A, x4) 归结得到 (12) Hate(B, C)

第2步:
(6) ~Rich(x6, A) | ~Kill(x6, A) 和 (7) Hate(B, x3) | Rich(x3, A) 归结得到 (18) Hate(B, x3) | ~Kill(x3, A)

第3步:
(0) Hate(A, A) 和 (3) Hate(B, x4) | ~Hate(A, x4) 归结得到 (11) Hate(B, A)

第4步:
(9) ~Hate(x5, C) | ~Hate(x5, A) | ~Hate(x5, B) 和 (11) Hate(B, A) 归结得到 (19) ~Hate(B, B) | ~Hate(B, C)

第5步:
(18) Hate(B, x3) | ~Kill(x3, A) 和 (19) ~Hate(B, B) | ~Hate(B, C) 归结得到 (26) ~Hate(B, C) | ~Kill(B, A)

第6步:
(12) Hate(B, C) 和 (26) ~Hate(B, C) | ~Kill(B, A) 归结得到 (49) ~Kill(B, A)

第7步:
(4) Hate(x1, A) | ~Kill(x1, A) 和 (5) Hate(C, x2) | ~Hate(A, x2) 归结得到 (14) ~Hate(A, A) | ~Kill(C, A)

第8步:
(0) Hate(A, A) 和 (14) ~Hate(A, A) | ~Kill(C, A) 归结得到 (21) ~Kill(C, A)

第9步:
(8) Kill(C, A) | Kill(A, A) | Kill(B, A) 和 (21) ~Kill(C, A) 归结得到 (38) Kill(B, A) | Kill(A, A)

第10步:
(10) ANSWER(u) | ~Kill(u, A) 和 (38) Kill(B, A) | Kill(A, A) 归结得到 (91) Kill(B, A) | ANSWER(A)

第11步:
(49) ~Kill(B, A) 和 (91) Kill(B, A) | ANSWER(A) 归结得到 (222) ANSWER(A)

归结成功!
共计用时: 79ms

E:\VisualStudio\VisualStudio_project\Honor\Debug\Honor.exe (进程 23772) 已退出, 返回代码为: 0。
若要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

最终得到 A 是自杀(A 杀了 A)。

实验结论: 使用前向连接的归结原理进行问题求解, 推理过程具有完备性, 在基于充分事实的基础上一定能够求解出正确答案。但是在归结过程中会产生大量冗余子句, 陷入盲目性归结, 需要使用相应的手段进行限制。

四. 总结

(1) 实验中存在的问题及解决方案

问题 1: 原始程序效率低下

解决方案: 原始程序通过遍历子句集来判断新生成的子句是否是重复出现, 并且在归结时对子句集中的所有子句都进行尝试, 因此使得时间复杂度迅速提高。但是通过分析发现, 可以使用 hash 将子句映射到 hash 表中, 通过 hash 结果来判断该子句是否之前出现过, 从而在低时间复杂度判断子句是否重复出现。并且, 每个子句只可能与和它具有互补文字的子句进行归结, 因此可以按照谓词公式, 通过将含有对应谓词公式的子句实现存储在“谓词公式”桶中, 每次归结时只从子句含有的谓词公式对应的一个或多个“谓词公式”桶中挑选子句进行归结, 大大提高了寻找归结子句的效率。

问题 2: 前向连接具有盲目性, 产生大量冗余子句

解决方案: 通过限制归结产生的子句的长度, 剪去冗余子句, 既节省了不必要的存储空间, 又使得最终有效推理路径上的归结能够尽早出现。同时提高了时间和空间效率。当不能归结出答案时逐渐放宽限制长度, 不会影响归结的完备性。在进行实验的过程中, 通过合适选取归结过程中允许产生的最大子句长度, 使得程序运行的时间从之前的一分钟左右降低到了 80ms。从中可以看出限制归结产生的子句的长度的威力。

问题 3: horn、point 类使用 string 类成员时会产生过高的空间开销, 占用大量内存, 影响运行速度。

解决方案: 使用小字符数组代替 string 类。由于变量、常量、函数的名称都不会太长(一般 1~3 个字符), 使用 string 类存储过于浪费空间, 因此只对谓词公式名称使用 string 类进行存储, 对变量、常量、函数名使用小字符数组进行存储。

(2) 后续改进方向

①前向连接固有弊端通过优化无法完全消除, 可以考虑使用后相连接代替前向连接, 更有目的地进行归结。

②可以使用递归方式实现更通用的合一置换算法, 包括对函数嵌套(如 $F(x, G(x))$)等的置换和发生检验。

③可以使用 C++ 的 STL 容器优化存储结构, 同时增强程序易读性。