一阶逻辑归结

摘 要

一阶逻辑归结是逻辑推理算法的基础，即智能体可以通过一阶逻辑子句集来判断要求证的真伪。

本次实验中，采用了前向链接的推理方法，即用原子语句替换、更新复杂子句来进行二元归结，验证与要求证是否一致。本实验使用 C++语言和 Qt 框架编写，完成了全部实验要求，并使用 UI 界面来帮助用户更好的使用和理解归结过程，该程序也可拓展用于其他类似问题的求解。

关键词：一阶逻辑归结，自动推理

装

订

线

First-order logic resolution

ABSTRACT

First-order logic resolution which means agent can judge correctness of goal clause by using first-order logical clause set, is the foundation of logical reasoning algorithm.

In this assignment, I use forward chaining reasoning algorithm that complex clause are deleted and replaced by using atomic clause and then compare consistence with goal clause. Besides, I accomplished this assignment completely using C++ language and Qt frame. I also add UI to help user better use it and understand the process. The assignment also can be used in other similar problems.

Key words： first-order logic resolution，automated reasoning

装

订

线

目 录

装 订 线

1 引 言

1.1 归结原理

归结（resolution）原理，在数理逻辑和自动定理证明中，是对于命题逻辑和一阶逻辑中的句子的推理规则。在命题逻辑中的归结规则是一个单一的有效的推理规则，从两个子句生成它们所蕴含的一个新的子句。归结规则接受包含互补的文字的两个子句 - 子句是文字的析取式，并生成带有除了互补的文字的所有文字的一个新子句。形式上，这里的 a_i 与 b_j 是互补的文字，其公式表示如图 1 所示：

$$\frac{a_1 \vee \ldots \vee a_{i-1} \vee a_i \vee a_{i+1} \vee \ldots \vee a_n, \quad b_1 \vee \ldots \vee b_{j-1} \vee b_j \vee b_{j+1} \vee \ldots \vee b_m}{a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_m}.$$

图 1 归结原理公式表示

归结规则生成的子句叫做两个输入子句的归结。

当两个子句包含多于一对的互补文字的时候，归结规则可以（独立的）应用到每个这种文字对上。但是，只有要消去的文字对可以去除：所有其他文字对仍保留在归结后的子句中。这一规则也被称为二元归结原则。

在一阶逻辑中，子句含有变量；为将归结原理应用于含有变量的子句，应找出一个置换，作用于给定的两个子句，使他们包括互补的文字，然后才能进行归结。

1.2 前向链接

前向链接算法：

从知识库的原子语句出发，在前向推理中应用假言推理规则，增加新的原子语句，直到不能进行任何推理

a、首先进行实例化的置换，得到 CNF，然后进行前向连接。

b、前向连接步骤：

①、扫描整个知识库，找到所有的实例化的事实（谓词），凡是有变量的都不是实例化的谓词

②、通过所有实例化的事实去置换所有的含有变量的事实（谓词），进行匹配，将可以进行置换的语句进行置换（实例化）

③、将新得到的实例化的事实加入到知识库中，并继续进行置换，直到得到结论。

2 实验概述

2.1 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。

2.2 实验内容

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。

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？

选用一种编程语言，在逻辑框架中利用归结原理进行求解。

要求界面显示每一步的求解过程。

撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT

2.3 本文所做的工作

本实验先根据题目要求将破案问题用九步法将一阶逻辑语言描述转换为子句（该过程人工完成，不在此赘述），编写了归结算法，对该问题进行求解，并可以在 UI 界面上进行人工处理后的子句集和要求证的输入，获得程序的使用方法。UI 界面可以显示完整的问题求解的过程与结果，对错误的输入予以提示。此外，整个项目设计不仅能解决并展示实验要求中的内容，还可以推广到类似的可以通过一阶逻辑推理解决的问题上去。

3　实验方案设计

3.1 总体设计思路与总体架构

将总体设计分为 UI 设计和内核设计两个部分，UI 部分负责与用户之间的交互和归结的过程与结果的展示，内核部分主要进行归结求解的过程。

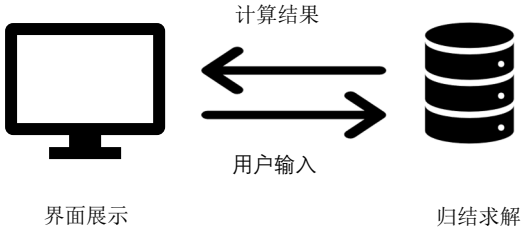


图 3.1 总体设计组成图

3.1.1 内核部分思路与框架

内核部分主要分为三个部分，输入字符的处理、逻辑归结、将过程结果转化为字符串，

输入字符的处理部分主要是对用户输入的字符串进行处理，判断子句、原子语句及其包含的函数名、变量名、常量名。逻辑归结部分则是利用 3.2.2 中涉及算法，利用原子语句对子句集中子句进行不断地替代和更新，验证是否能得到要求证，这也是本次课程设计最核心的部分。最后是将归结的过程与结果转化为字符串，以便于进一步的输出。

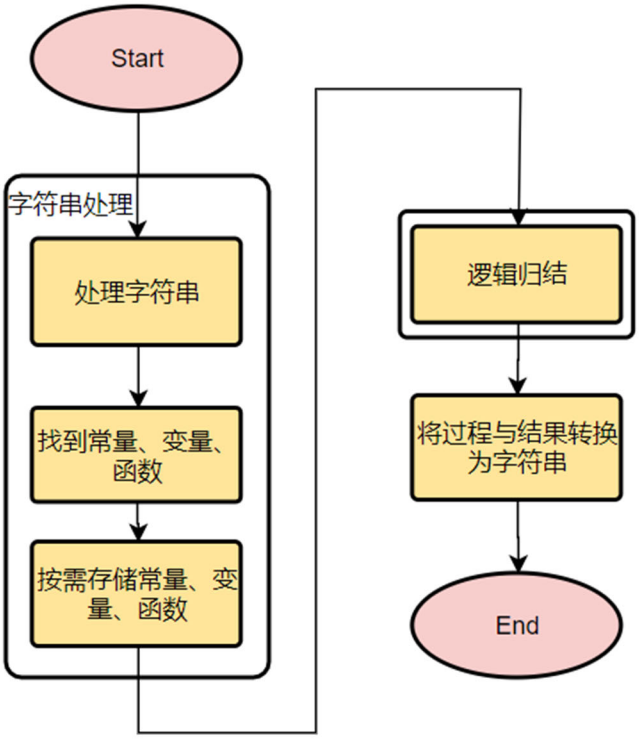


图 3.1.1 内核部分框架图

3.1.2 UI 部分思路与框架

UI 界面按功能分为菜单栏、输入部分、输出部分三大块。

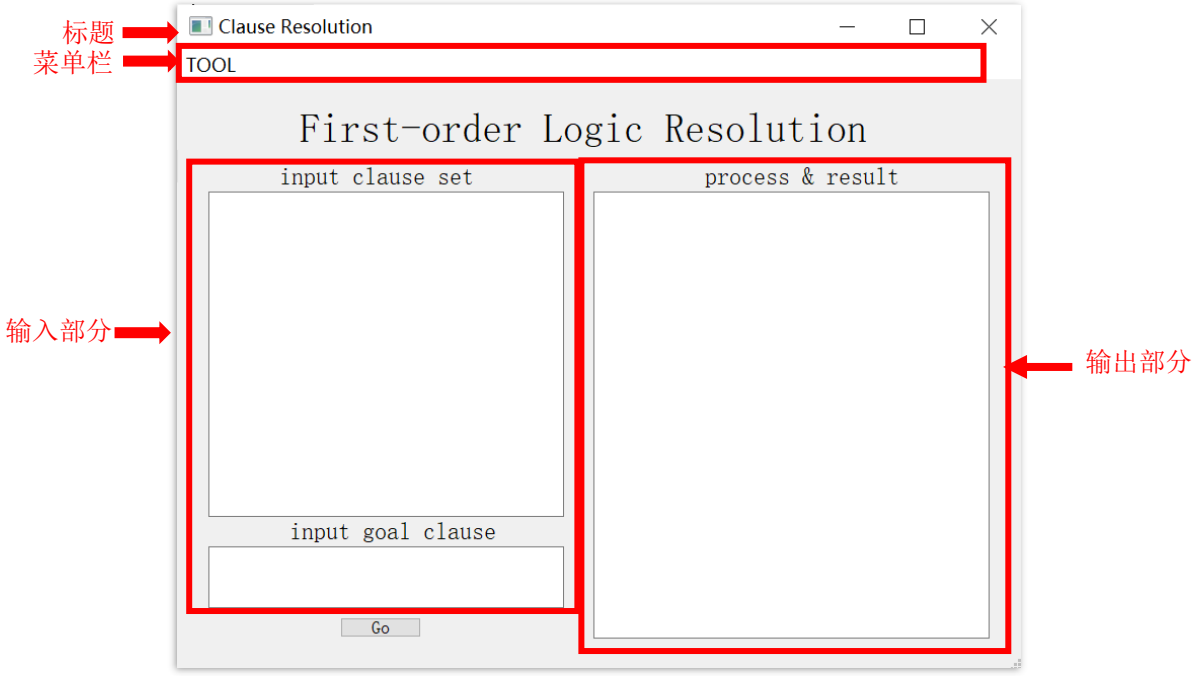


图 3.1.2.1 主界面展示

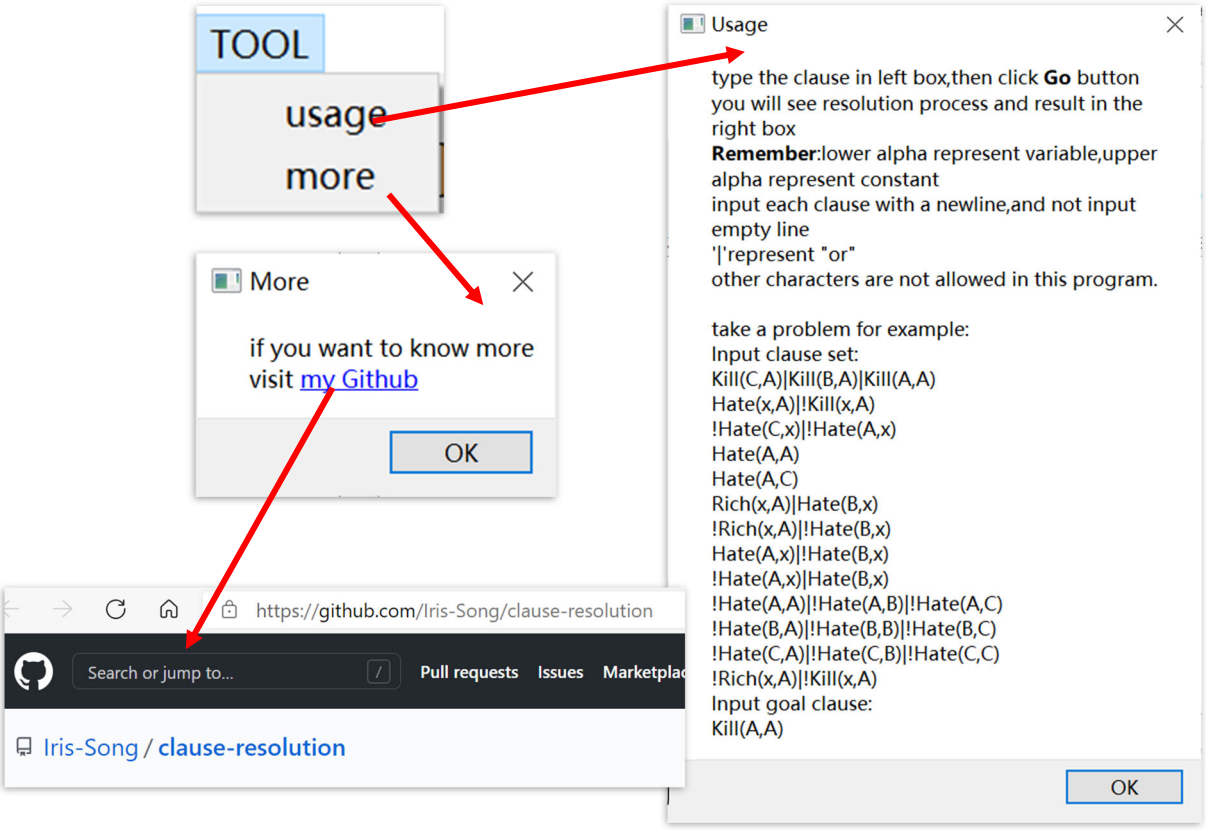


图 3.1.2.2 TOOL 下拉栏展示

装 订 线

可在输入部分填入对应子句集和要求证，点击 Go 按钮会开始运算，若为空或不符合输入要求会有对应提示。输出部分会显示内核部分的运算结果。

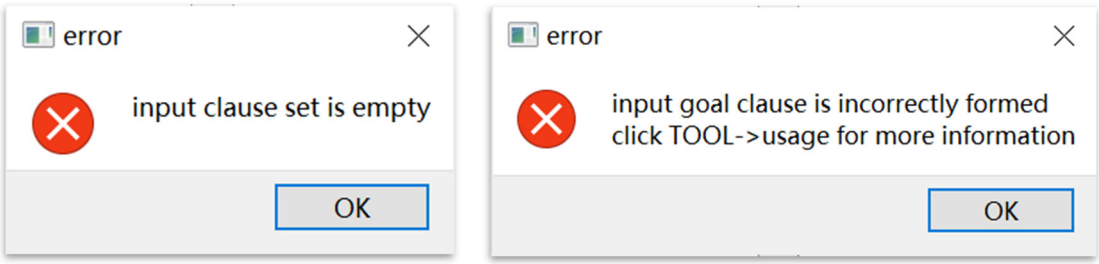


图 3.1.2.3 可能的输入错误提示 左：输入子句集为空 右：要求证格式不正确

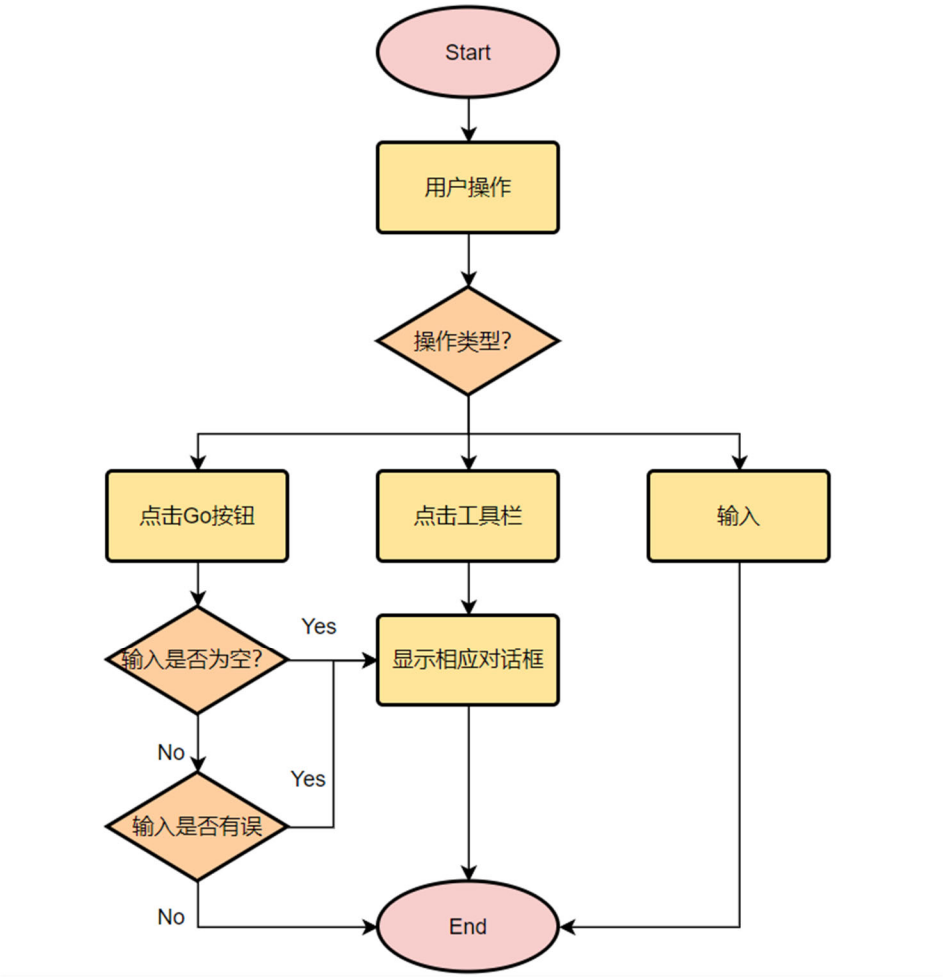


图 3.1.2.4 UI 部分流程图

3.2 核心算法及基本原理

3.2.1 输入字符处理

从 UI 中获得的是用户输入的字符串，程序会自动在字符串结尾加入'#'标识符，并以此为依据判断字符串是否结束。首先，需要先从字符串中获得子句。判断子句结束的依据是每句结尾的

换行符。如果子句中仅有换行符则予以忽略。然后从子句中获得原子语句（即不可再分的最小子句，比如 Kill(C,A)|Kill(B,A)|Kill(A,A)，可依次分为 Kill(C,A)、Kill(B,A)、Kill(A,A)三个原子语句）。再从原子语句中找出函数名、变量名、常量名。事先约定好变量名由一个及以上的小写字母组成，常量名由一个及以上的大写字母组成。判断函数名的标准是寻找原子语句'('前的子串，找变量名与常量名之间以','')'为分隔，若出现标识符为大小写混合等情况，则其不合要求，字符串处理失败。

以对子句集的处理为例，流程图如下：

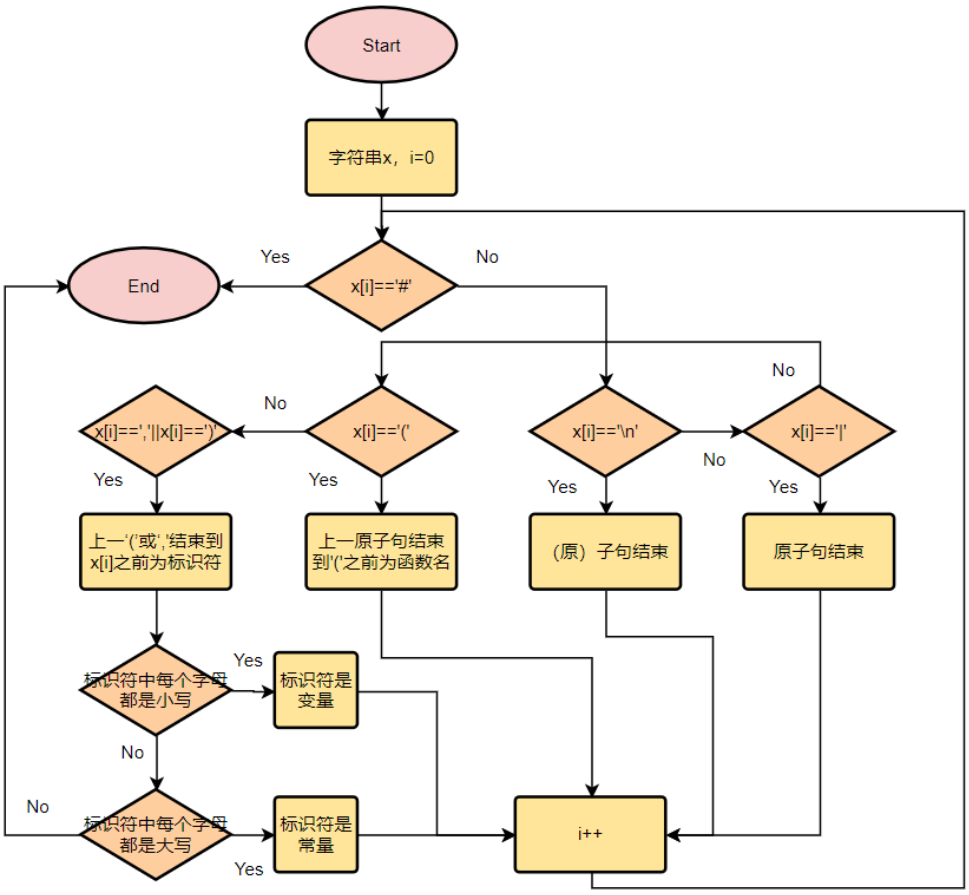


图 3.2.1 输入字符处理流程图

3.2.2 归结算法

使用了前向链接的方法。首先对子句集中的子句进行一一判断，如果存在原子语句且该原子语句与要求证相一致或相抵触，则对应返回成功或失败。否则进行逻辑归结。将子句集中的原子语句设为不可访问。如果子句集中的原子语句与另一子句（复杂子句）中的某个原子语句除了变量和符号（指整个原子语句的符号）外均相同，则用不可访问的原子语句去替代更新可访问的子句。

具体更新替换的方法如下：

如果更新可访问的子句中的标识符是变量，则用原子语句中对应的常量去替换变量。最后

将替换后的子句加入子句集。如果在替换过程中出现了标识符为空的情况，则输入的子句集本身逻辑有误。依据更新替换结果，返回对应的状态。

若在更新替换过程中出现逻辑错误或得到要求证结论，则直接返回最终结果。若将所有子句都已变为原子子句（子句集中所有子句都已变为不可访问状态），仍无法归结出要求证，则归结失败。

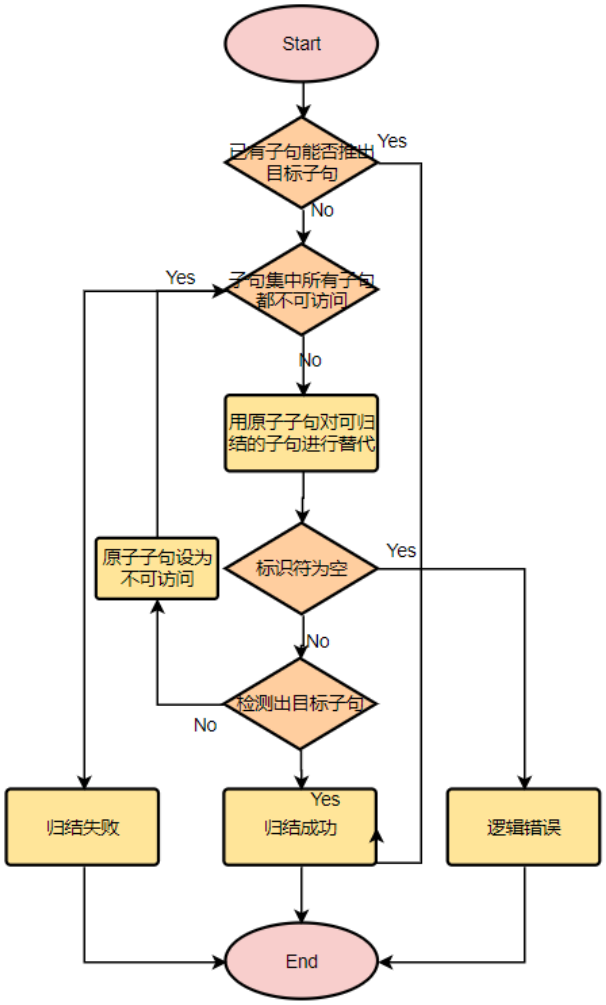


图 3.2.2 归结算法流程图

3.3 模块设计

3.3.1 内核模块设计

A. 类设计

先定义了 constant_name、variable_name、function_name 三个 string 型的 vector 分别存储常量名、变量名、函数名。

定义了 identifier 类，代表标识符。type 是标识符的类型，id 则是该类型标识符的名称在所对应 vector (constant_name、variable_name、function_name) 中的编号。

```
enum identifier_type {CONSTANT,VARIABLE};
```

```
class identifier
{
public:
    identifier_type type;//常量还是变量
    int id;//vector 中的标号
};
```

定义了 atomic 类，表示原子式，element vector 顺序存储原子式中所有的标识符；function_id 记录原子式在 function_name 的下标，bool 变量 positive 记录原子式前面是否有非

```
class atomic{
public:
    std::vector<identifier> element;
    int function_id;
    bool positive=true;//原子式前面有非，则为 false，否则为 true
};
```

定义了 clause 类，表示子句，mother、father 分别为该子句产生的两个来源(表示在子句集中的下标)，将其初始化为-1;bool 变量 valid 表示子句是否可访问，初始化为 true; bool 变量 key 表示若能推得要求证，其是否是推得结论的关键子句，初始化为 false。

```
class clause {
public:
    std::vector<atomic> element;//按顺序存储所有以或连接的原子语句
    int mother=-1;//来源 1，没有则置为-1
    int father=-1;//来源 2，没有则置为-1
    bool valid=true;// 子句集是否可访问
    bool key=false;//是否式关键子句
};
```

B. 输入字符串处理

见 3.2.1。

C. 归结

见 3.2.2。

D. 过程与结果转换为字符串

对每个子句进行标号，以子句为单位输出（使用 display_clause 函数）输入的子句集。若不能在子句集中直接得到要求证，则对其后放入子句集中的归结子句进行编号后输出。如果最后能够成功归结出要求证，则递归找出关键子句，对关键路径处的子句 key 变量设为 true。使用<Qstring>头文件，对关键路径处的子句用蓝色字体输出。根据最后归结结果，加入结果说明的提示。

3.3.2 UI 模块设计

UI 模块主要通过 Qt 框架实现。

与内核模块之间的联系，详见 3.3.3。

A. 输入输出的实现

使用了 Qt 自带的 textEdit 实现子句集和要求证的输入；textBrowser 实现结果和过程字符串的展示。

B. 菜单栏及对话框的实现

利用信号-槽结构实现。使用了 QmessageBox 等头文件，使用到了简单的 html 语言来实现对话框中的提示和网页的跳转。

3.3.3 内核部分与 UI 部分的联系

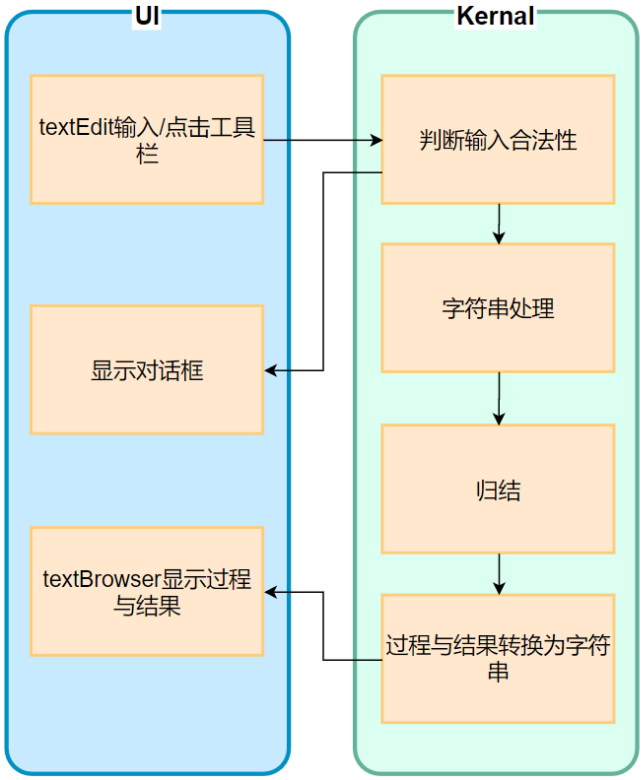


图 3.3.3 内核部分与 UI 部分的联系

3.4 创新内容或优化算法

A.可能存在的输入处理

由于无法预测用户可能的全部输入，需要对常见的可能输入做处理。比如用户可能会输入多个空换行，则对空换行抛弃不读；无法得知子句集输入的结尾，程序会在字符串结尾自动加上换行符作为最后一个子句结束的标志，加入'#'作为整个子句集的结束标志。如图 3.4.1。对于要求证的输入，由于要求证有且仅有一个，比较好处理，情况与子句集的处理类似。

B. 检查输入子句集

在将问题不仅局限于实验内容所描述时，有的问题的结果无需归结，子句集中就已直接声明，此时就无需进入归结过程即可直接得出结论，这也大大提高了程序的运行效率。如图 3.4.2 的简单子句集，仅凭输入子句集便能得出。

C. 关键子句的标注

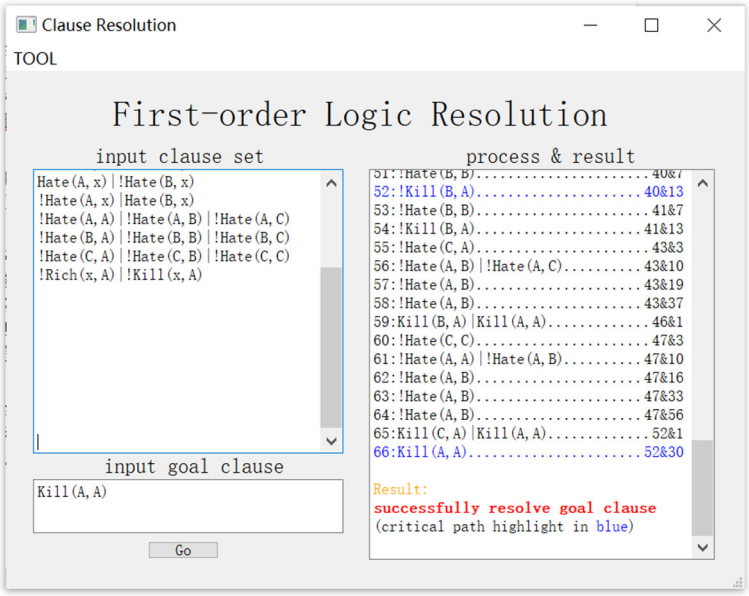
在归结成功的前提下，我对归结的关键路径进行了标注，并在显示时，将其用特殊的颜色显示，使用户能直观的了解到归结的全过程。

具体的寻找关键子句的方法采用了递归。由归结过程可以得知，最后加入子句集的子句为最后转换出的要求证，从子句集的最后一个子句开始，递归地寻找它的双亲（由双亲可以推得该子

句），直到没有双亲为止，递归结束。在显示时，将标记为关键路径的子句用蓝色输出。

D.UI 界面设计

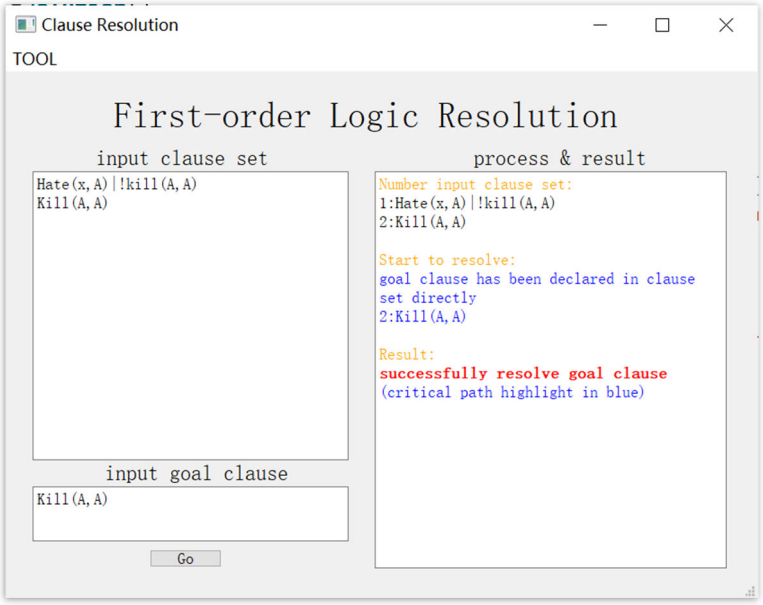
使用了 Qt 为框架，在归结过程中能对过程和结果使用不同的颜色标注，使得整个程序更加



清晰易读。增加了工具栏和相对应的提示，可以使程序开发者以外的人更好的使用程序。

图 3.4.1 加入多个换行符后仍能得到正确结果

图 3.4.2 直接从输入子句集中推得要求证



4 实验过程

4.1 环境说明

硬件配置：Intel Core i5-10210U CPU

操作系统：64 位 Windows10

开发语言：C++

开发环境：Qt Creator (Qt 5.12.10 MSVC2017 64bit)

编译器：Microsoft Visual C++ Compiler 16.5.30104.148(x86_amd64)

调试器：CDB Engine

4.2 源代码文件清单及主要函数清单

4.2.1 头文件

A. head.h

引用了 C++中的头文件库，定义整个程序中使用到的变量和类。部分已在 3.3.1 A 中介绍。声明了 clause 型的 vector，clause_set 来存储子句集中的各个子句。使用了 Qt 框架中的 QString 型，result 来表示将要输出的字符串。

B. mainwindow.h

包含与 UI 界面有关的头文件，声明了 MainWindow 类，该类中包含与界面有关的变量与函数(包括 slot 函数)。

C. ui_mainwindow.h

声明并实现了 UI 界面。

4.2.2 源文件

A. main.cpp

包含 mainwindow.h

main 函数所在，显示主程序

B. mainwindow.cpp

界面源文件，包含 mainwindow.h, ui_mainwindow.h, read.cpp, solution.cpp

a. MainWindow 类的构造函数和析构函数

构造函数与析构函数分别对 UI 界面进行建立和删除。

b. void MainWindow:: init ()

对进行 head.h 中的变量和 UI 中的部件进行初始化，。

c. void MainWindow:: on_GoButton_clicked()

Go 按钮的信号槽函数，是整个程序的核心函数，会调用 init ()函数对变量进行初始化。然后获取输入框中的内容并判断其是否为空。若不为空，则判断输入是否合法。若合法，则继续进

行归结，并输出归结结果。

d. void MainWindow:: UsageClicked()

TOOL 菜单下拉栏中关于该程序使用的槽函数。会有消息框介绍相关信息。

e. void MainWindow:: MoreClicked()

TOOL 菜单下拉栏中关于更多情况的槽函数。会有消息框介绍相关信息。

C. read.cpp

字符串处理源文件。包含 head.h

a. bool find_func_id(std::string present_function_name, int&id)

bool find_identifier_id(std::string present_identifier_name, identifier& present_identifier)

根据函数名/标识符的名字查找在对应 vector 中的下标，返回查找是否成功

b. bool add_identifier_id(std::string present_identifier_name, identifier& present_identifier)

在已知标识符不在 identifier 的情况下，加入 identifier 中，返回加入是否成功内

c. bool read_one_atomic(clause& present_clause, std::string atomic_str)

读入一个原子语句，得到 clause 类，返回读入是否成功

d. bool read_clause(std::string x)

读入子句集中的子句，调用 read_one_atomic 函数。返回读入是否成功。

e. bool read_dst(std::string x)

读入要求证，调用 read_one_atomic 函数。返回读入是否成功。

D. solution.cpp

归结源文件，包含 head.h。

a. bool exp_v_pos_equal (atomic atomic1, atomic atomic2)

判断两个原子语句是否除变量和真假外全部一致。

bool exp_pos_equal (atomic atomic1, atomic atomic2)

判断两个原子语句是否除真假外全部一致。

b. bool is_child(int clause1, int clause2)

判断 clause1 是否是 clause2 的孩子节点。

c. int replaced(int fact, int replaced_clause, int replaced _atomic)

fact、replaced_clause、replaced _atomic 都是所对应的 vector 下标，用 fact 原子语句去替换更新 replaced_clause 中的 replaced _atomic。将替换后的子句放入子句集，并判断能否判断出要求证，返回归结状态。

int substitude(int num)

调用 replaced 用 clause[num]替换 clause_set 中的子句，返回归结状态。

int resolution()

调用 substitude，遍历所有的子句集进行归结，返回归结状态。

三个函数一起构成了归结算法，归结算法详见 3.2.2.

d. void find_critical_path(clause &present_clause)

递归寻找关键子句，详见 3.4. B。

e. int count_clause_len(clause present_clause)

计算并返回 present_clause 的字符长度。

f. void display_resolution(int num,int state)

展示归结过程。对输入子句集和归结过程中产生的子句进行标号，若 state 状态为归结成功，则调用 find_critical_path 找到关键子句，输出整个归结过程和结果提示。

4.2.3 表单文件

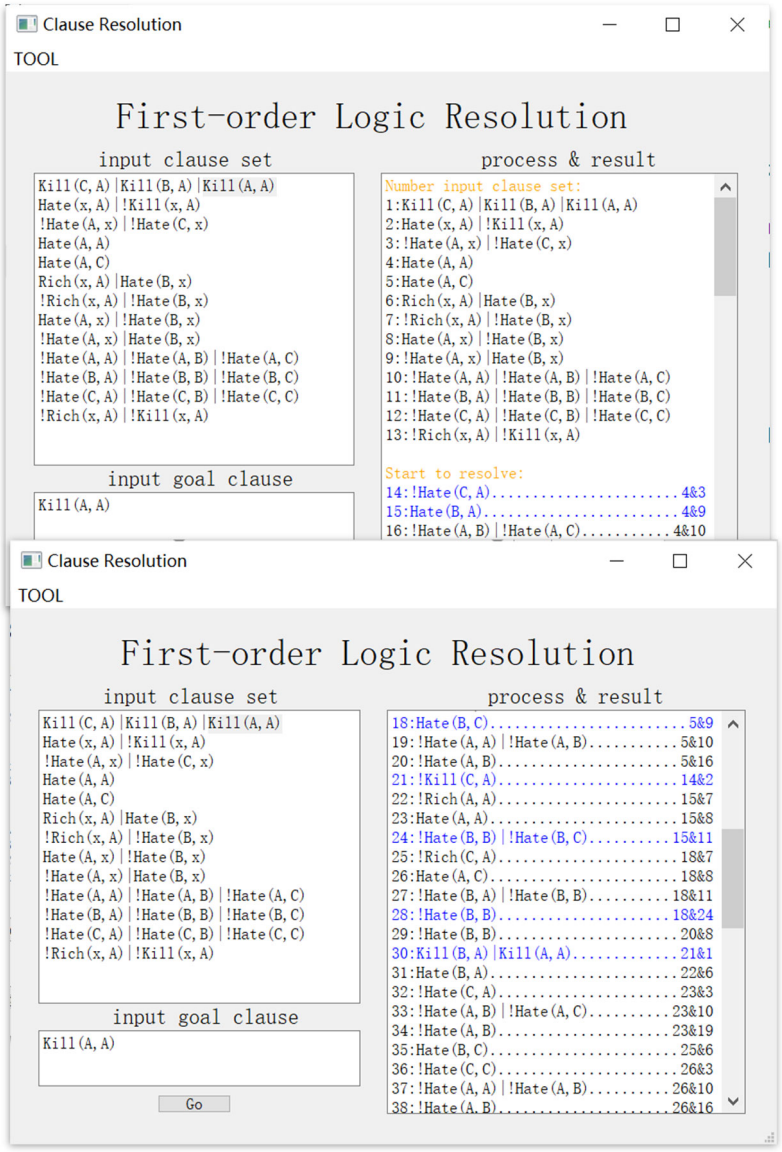
A. mainwindow.ui

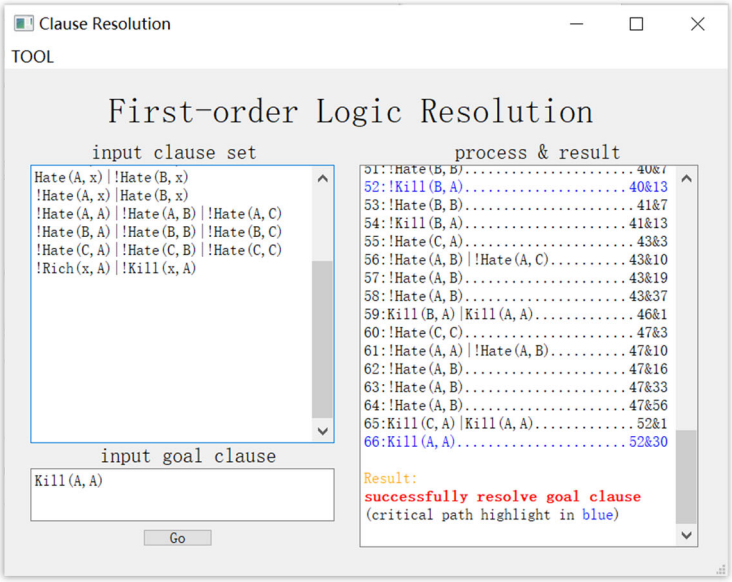
借助 Qt 框架，图形化界面实现 UI 窗口。

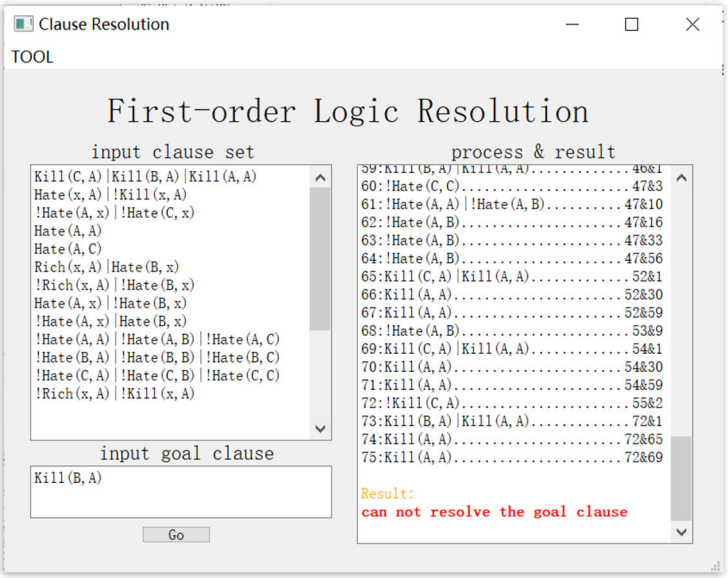
4.3 实验结果展示

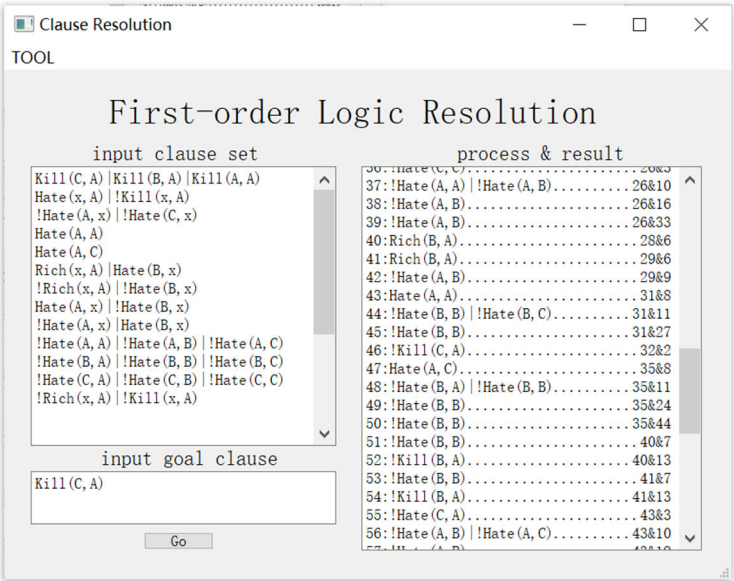
4.3.1 UI 界面展示

将文字语言转换为一阶逻辑表达式。Kill(x,A)表示 x 杀了 A。Hate(x,y)表示 x 恨 y。Rich(x,y)表示 x 比 y 富有。输入 Kill(A,A)作为要求证发现其能得出归结正确，Kill(B,A) 、Kill(C,A)则会归结错误。




Clause Resolution
TOOL
First-order Logic Resolution
input clause set
Hate(A, x)|!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)
!Rich(x, A)|!Kill(x, A)
input goal clause
Kill(A, A)
Go
process & result
52:!Kill(B, A).....40&13
53:!Hate(B, B).....41&7
54:!Kill(B, A).....41&13
55:!Hate(C, A).....43&3
56:!Hate(A, B)|!Hate(A, C).....43&10
57:!Hate(A, B).....43&19
58:!Hate(A, B).....43&37
59:Kill(B, A)|Kill(A, A).....46&1
60:!Hate(C, C).....47&3
61:!Hate(A, A)|!Hate(A, B).....47&10
62:!Hate(A, B).....47&16
63:!Hate(A, B).....47&33
64:!Hate(A, B).....47&56
65:Kill(C, A)|Kill(A, A).....52&1
66:Kill(A, A).....52&30
Result:
successfully resolve goal clause
(critical path highlight in blue)


Clause Resolution
TOOL
First-order Logic Resolution
input clause set
Kill(C, A)|Kill(B, A)|Kill(A, A)
Hate(x, A)|!Kill(x, A)
!Hate(A, x)|!Hate(C, x)
Hate(A, A)
Hate(A, C)
Rich(x, A)|Hate(B, x)
!Rich(x, A)|!Hate(B, x)
Hate(A, x)|!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)
!Rich(x, A)|!Kill(x, A)
input goal clause
Kill(B, A)
Go
process & result
60:!Hate(C, C).....47&3
61:!Hate(A, A)|!Hate(A, B).....47&10
62:!Hate(A, B).....47&16
63:!Hate(A, B).....47&33
64:!Hate(A, B).....47&56
65:Kill(C, A)|Kill(A, A).....52&1
66:Kill(A, A).....52&30
67:Kill(A, A).....52&59
68:!Hate(A, B).....53&9
69:Kill(C, A)|Kill(A, A).....54&1
70:Kill(A, A).....54&30
71:Kill(A, A).....54&59
72:!Kill(C, A).....55&2
73:Kill(B, A)|Kill(A, A).....72&1
74:Kill(A, A).....72&65
75:Kill(A, A).....72&69
Result:
can not resolve the goal clause


Clause Resolution
TOOL
First-order Logic Resolution
input clause set
Kill(C, A)|Kill(B, A)|Kill(A, A)
Hate(x, A)|!Kill(x, A)
!Hate(A, x)|!Hate(C, x)
Hate(A, A)
Hate(A, C)
Rich(x, A)|Hate(B, x)
!Rich(x, A)|!Hate(B, x)
Hate(A, x)|!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)
!Rich(x, A)|!Kill(x, A)
input goal clause
Kill(C, A)
Go
process & result
37:!Hate(A, A)|!Hate(A, B).....26&10
38:!Hate(A, B).....26&16
39:!Hate(A, B).....26&33
40:Rich(B, A).....28&6
41:Rich(B, A).....29&6
42:!Hate(A, B).....29&9
43:Hate(A, A).....31&8
44:!Hate(B, B)|!Hate(B, C).....31&11
45:!Hate(B, B).....31&27
46:!Kill(C, A).....32&2
47:Hate(A, C).....35&8
48:!Hate(B, A)|!Hate(B, B).....35&11
49:!Hate(B, B).....35&24
50:!Hate(B, B).....35&44
51:!Hate(B, B).....40&7
52:!Kill(B, A).....40&13
53:!Hate(B, B).....41&7
54:!Kill(B, A).....41&13
55:!Hate(C, A).....43&3
56:!Hate(A, B)|!Hate(A, C).....43&10

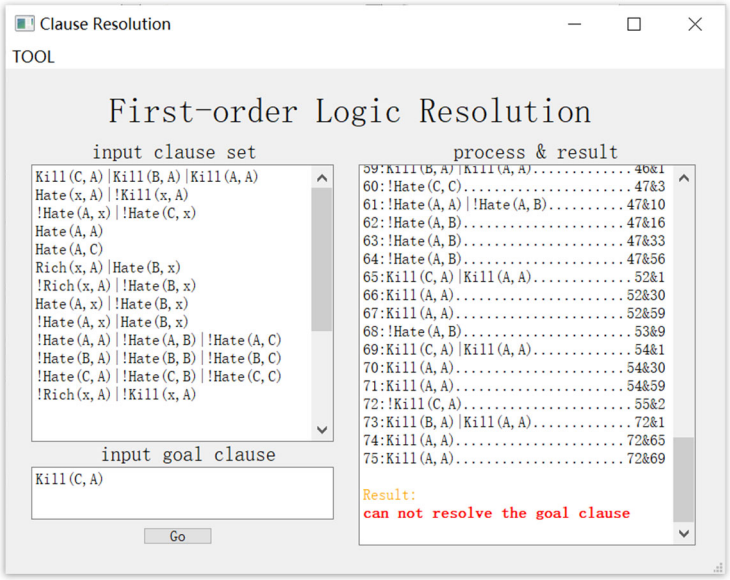


图 4.3.1 结果展示

4.3.2 结果分析

经过程序推理得到的 4.3.1 结果，不难看出，谋杀者是 A。

5 总结

参考文献

[1] Jasmin Blanchette, Mark Summerfield. C++ GUI Qt 4 编程. 北京:电子工业出版社, 2018，2000(2):5-8.

[2] https://zh.wikipedia.org/wiki/%E5%BD%92%E7%BB%93%E5%8E%9F%E7%90%86

[3] Stuart Russell, Peter Norvig. Artificial intelligence: a modern approach 北京:人民邮电出版社, 2002

装

订

线