一阶逻辑归结
求解破案问题

学　　号 [redacted]

姓　　名 [redacted]

专　　业 计算机科学与技术专业

授课老师 [redacted]

目 录

1 实验概述.....
1.1 实验目的.....
1.2 实验内容.....
2 实验方案设计.....
2.1 总体设计思路与总体框架.....
2.1.1 总体设计思路.....
2.1.2 总体框架.....
2.2 核心算法及基本原理.....
2.2.1 核心算法.....
2.2.2 实现基本原理.....
2.3 模块设计.....
2.4 其他创新内容或优化算法.....
3 实验过程.....
3.1 环境说明.....
3.2 源代码文件清单.....
3.2.1 源代码清单.....
3.2.2 主要函数清单.....
3.3 实验结果展示.....
3.3.1 界面结果展示.....
3.3.2 运行结果展示.....
3.4 实验结论.....
4 总结.....
4.1 实验中存在的问题及解决方案.....
4.2 心得体会.....
4.3 后续改进方向.....
4.4 总结.....
参考文献.....
附录.....

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。

1.2 实验内容

1.2.1 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？

1.2.2 要求

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，在逻辑框架中利归结原理进行求解。
3. 要求界面显示每一步的求解过程。
4. 撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT。

2 实验方案设计

2.1 总体设计思路与总体框架

2.1.1 总体设计思路

本程序的总体设计思路是：

项目首先设计一个 clauseResolution 类对一阶逻辑进行归结处理。对外接口设计部分：clauseInput 函数按照预设数据结构对子句集进行存储；resolution 函数执行子句归结任务；traceBack 函数对归结结果进行回溯，得到最优归结路径；allShow\bestShow\ansShow 函数分别打印归结结果\最优归结路径以及结果最终结果。外部执行方面，只需要调用对应的函数即可实现子句归结任务。

在项目交互方面，设计一个 clauseUI 类进行实现。接口定义方面：bgShow 函数对交互界面进行处理显示；orderChoose 函数读取按键信息，返回对应操作指令。

综合一阶逻辑实现类和 UI 界面类，便可实现项目所要求的对破案问题的求解、展示和交互任务。总体设计清晰简介，各个类的任务分配较为合理，实现过程简单高效。

程序设计安排再 4 个 cpp 文件中实现，分别为：

logicalReasoning_head.h：存储整个项目类、结构体、函数、宏定义的声明，作为项目的头文件使用。

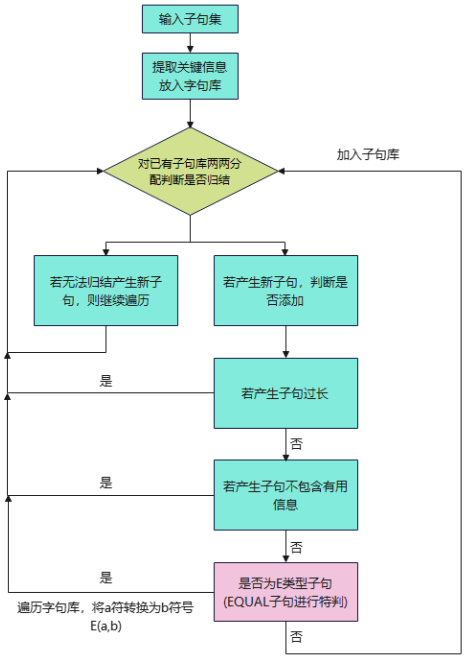
logicalReasoning_kernel.cpp：内含 clauseResolution 类的具体实现函数，实现一阶逻辑归结的内部核心算法的求解。还包含部分整个实验所需工具函数的具体实现。

logicalReasoning_ui.cpp：内含 clauseUI 类的具体实现函数，实现破案问题的可视化界面输入输出。

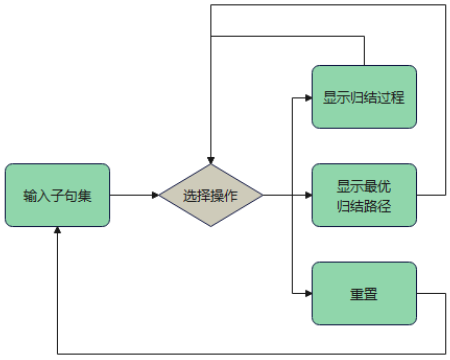
logicalReasoning_main.cpp：程序主函数所在地，调用各个类进行对破案问题进行子句归结求解。

2.1.2 总体架构

程序核心 ---- 子句归结实现的总体架构：



根据此总体程序实现过程，以期望实现的程序运行流程如下：



2.2 核心算法及基本原理

2.2.1 核心算法

本实验的核心算法即归结原理。

其归结过程为：对于两个已经完成变量标准化没有共享变量的两个子句，如果包含互补文字则可对它们进行归结。即如果一个命题文字是另一个命题文字的否定式，则这两个命题文字是互补的；如果一个一阶逻辑文字能和另一个一阶逻辑文字的否定式合一，则这两个一阶逻辑文字是互补的。由此可得到：

$$\frac{l_1 \vee l_2 \vee \ldots \vee l_k \quad m_1 \vee m_2 \vee \ldots \vee m_n}{SUBST(\theta, l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{i-1} \vee m_{i+1} \vee \ldots \vee m_n)}$$

其中 UNITY(li,¬ mj)= θ 。换言之，归结原理的本质就是通过合一置换消除互补文字，从而形成一个新子句的过程。

该过程被称为二元归结规则，它正好对两个文字进行归结。二元归结本身不能产生完备的推理过程。全归结规则对每个可合一的子句中的文字子句进行归结。

本次实验综合使用置换合一规则及归结规则进行解题。

首先将已知条件用谓词公式表示出来，并转化成子句集。此时需要根据已知条件定义谓词，并将事实用谓词公式表示出来，并将它们化成子句集。此时，关键需要将问题用谓词公式表示出来，使用已定义谓词+A(Answer)谓词对问题进行转化，添加进入子集中。对现有的所有子集进行归结，当得到 A 谓词的最终结果且内含已知值，归结结束，A 谓词所含结果即为所求。

算法核心使用归结规则+置换\合一规则，关键在于多次使用以上规则，直至 A(Answer)谓词中置换出已知固定值，此值即为问题所需求得的答案。

2.2.2 实现基本原理

本题在于实现归结+置换+合一，因此，综合考虑以上内容，可以得到算法基本实现过程如下：

Ⅰ.读取输入子集，存储至子集库中
Ⅱ.遍历选择子集库中两个子句进行归结尝试，能进行归结需要两子句中存在以下的表达式
  a. 谓词种类一致，且一个为非；一个为正
  b. 谓词中元素若均为常量，则两变量需要一致
  c. 谓词中元素若均为变量，则满足可归结
  d. 谓词中一个元素为变量，一个元素为常量或者函数，则可使用合一置换规则进行归结
Ⅲ.若满足Ⅱ中条件则进行归结，否则回到Ⅱ继续执行下一对尝试
  a. 执行归结，获得置换合一后的新语句
  b. 判断语句是否过长
  c. 判断语句是否存在包容归结
  若不满足以上条件回到Ⅱ继续执行；反之，执行下一语句
Ⅳ. 判断语句是否为结束语句(逻辑 A)。若是，则结束程序，得到结果；反之添加进子集库中，重复执行语句Ⅱ

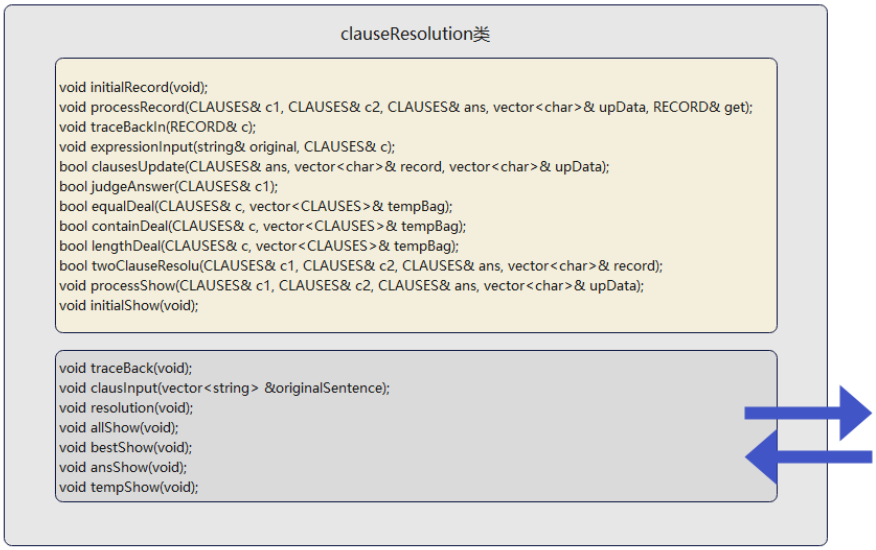
按照以上实现过程，便可以实现一阶逻辑的子句归结任务。

2.3 模块设计

①clauseResolution 模块

```cpp
// @function : 子句归结执行内核
class clauseResolution {
protected:

    // 内部参数
    CLAUSES answer;                                         // 最终结果
    int countNum = 0;                                       // 用于记录转换个数
    vector<char>expressionType;                             // 该推断问题中表达式的所有种类 --> 'A'-'Z'代替
    vector<CLAUSES>clauseBag;                               // 存储所有转化后的子句集

    // 可视化结果存储
    vector<RECORD>initialList;                              // 初始化数据存储
    vector<RECORD>recordList;                               // 过程数据存储
    vector<RECORD>bestPath;                                 // 最优归结路线

    // 内部函数
    void initialRecord(void);                                                                       // 初始逻辑记录
    void processRecord(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData, RECORD& get); // 归结过程记录
    void traceBackIn(RECORD& c);                                                                    // 寻找最佳归结路线内部递归函数
    void expressionInput(string& original, CLAUSES& c);                                             // 将一个字符串转换成
    bool clausesUpdate(CLAUSES& ans, vector<char>& record, vector<char>& upData);                   // 子句更新
    bool judgeAnswer(CLAUSES& c1);                                                                  // 判断是否归结到终点
    bool equalDeal(CLAUSES& c, vector<CLAUSES>& tempBag);                                           // 等式归结[若为E(f(A),B)则将f(A)替换为B]
    bool containDeal(CLAUSES& c, vector<CLAUSES>& tempBag);                                         // 包容归结
    bool lengthDeal(CLAUSES& c, vector<CLAUSES>& tempBag);                                          // 长度限制
    bool twoClauseResolu(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& record);             // 两个子句归结
    void processShow(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData);                 // 控制台打印过程
    void initialShow(void);                                                                         // 控制台打印初始过程


public:
    // 外部函数
    void traceBack(void);                                   // 寻找最佳归结路线
    void clausInput(vector<string> &originalSentence);      // 将输入子句转化为易计算结构子句
    void resolution(void);                                  // 执行子句归结
    void allShow(void);                                     // 展示所有归结过程
    void bestShow(void);                                    // 展示最优归结过程
    void ansShow(void);                                     // 显示最终结果

    void tempShow(void);                                    // for test
};
```

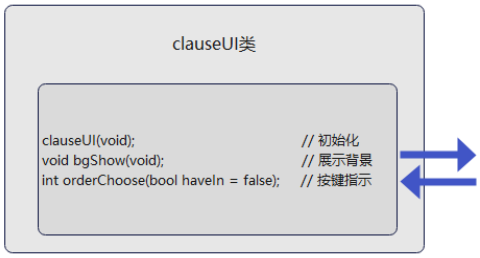
该模块的整体模块设计为：



②clauseUI 模块

```
// @function : 字句归结UI界面
class clauseUI {
protected:
    // 界面布局结构体定义
    SURFACE backArea;
    SURFACE inputArea;
    SURFACE showallArea;
    SURFACE showbestArea;
    SURFACE restartArea;

    // 界面图片调用定义
    IMAGE background;
    IMAGE input;
    IMAGE inputActive;
    IMAGE showAll;
    IMAGE showAllActive;
    IMAGE showBest;
    IMAGE showBestActive;
    IMAGE restart;
    IMAGE restartActive;

public:
    clauseUI(void);                                  // 初始化
    void bgShow(void);                               // 展示背景
    int orderChoose(bool haveIn = false);            // 按键指示

};
```

该模块的整理模块设计为：



2.4 其他创新内容或优化算法

2.4.1 E 关键字逻辑实现

E 关键字指的是破案问题中规定的逻辑表示：即 E(a,b)表示 a 和 b 是相等的。而之所以这个函数要单独拿出来处理，就是因为这种相等关系，使得子句集中的其他字句可以以该函数为推理基础，对文字的参数列表进行替换。

以该问题为例，存在如下两个推理出的子句：

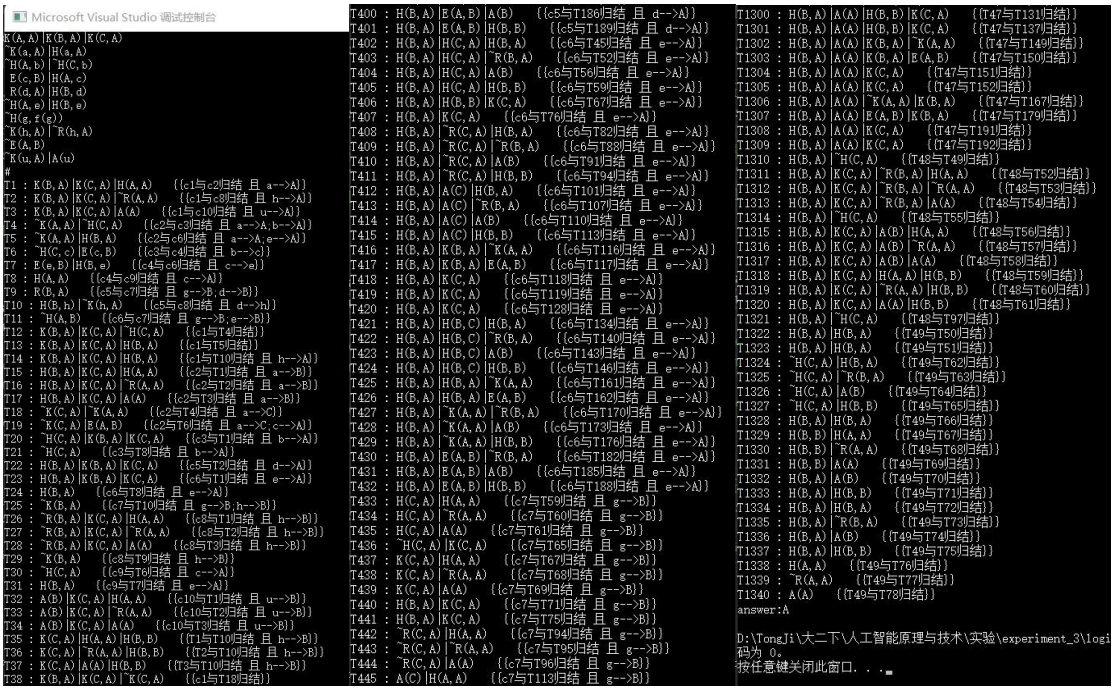
T20:R(f(B),A)

T25:E(f(B),B)

根据这两个子句可知，f(B)与 B 是等价的，所以应当对 T20 子句进行一次变量替换得到替换后的子句 R(B,A)。经过程序调试发现，如果在程序运行过程中不考虑该置换，则无法归结出结果。因此，当出现 E 语句时，需要对全局已存在子句进行遍历替换，这是有必要的。由于我们已经执行了 E 逻辑的功能，若再送入子集库中，显然对最终的结果无任何利处，还将损害程序执行效率。

因此，无需将 E 逻辑语句再送入子集库中。

综上，处理 E 逻辑语句的做法是：遍历所有语句，进行替换，并无需再将其送入子集库中。执行上述操作后，程序能正确运行，结果如下图。



由上图执行结果可知：总共需要执行 1340 次归结，最终结果得到凶手为 A。

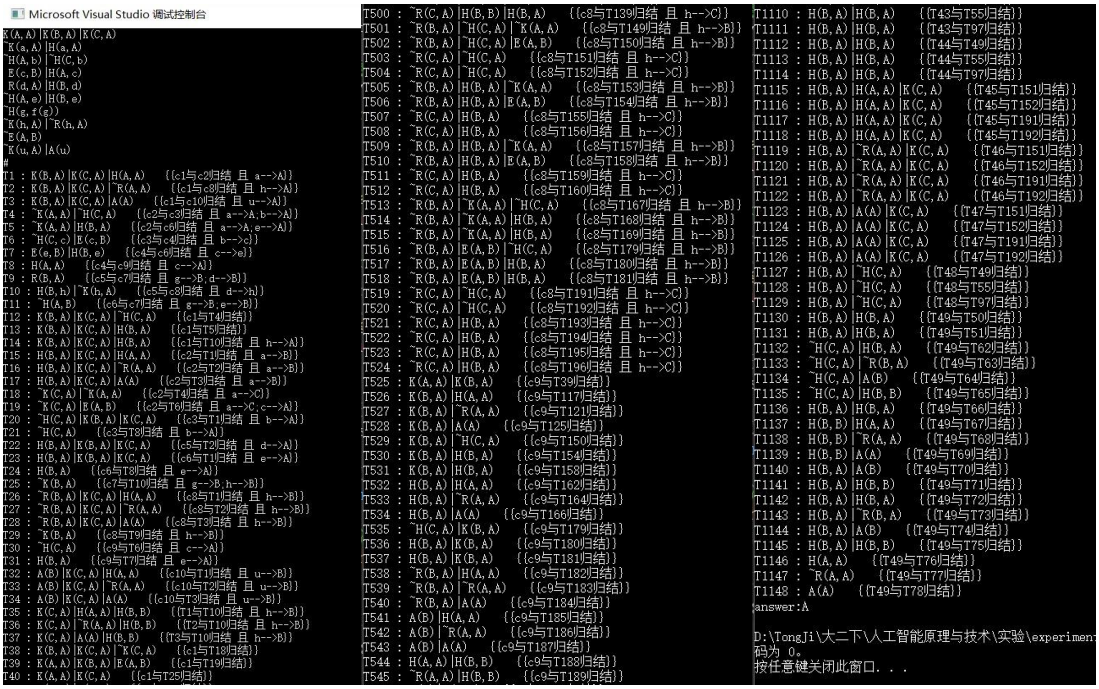
2.4.2 长度限制策略

长度限制策略是指，删除归结结果语句中的部分语句，这些被删除的语句往往长度大于原始子句集中的最大长度，而这些语句包含的内容常常是多余的。

所以我们可以在一开始就对试图进入子集库的子句 p 进行一次判断，如果其长度大于初始子句集中的最大长度，那么就可认为该子句集是“低效”的，直接拒绝其入子集库请求即可。

这个过程看似较为“鲁莽”，但其实仔细想来，归结的最终目的是要归结出长度为 1 的 A 逻辑子句，那么一般的归结过程就应该是新生成的子句长度越来越小，直至生成空子句——而在这一过程中我们的确应当减少对过长子句的考虑，以减少搜索时间。

这一优化策略的确是不完备的，有一定概率会使得归结结果不准确，但正如线性归结策略一样，这种情况发生的概率较小，且该情况不符合绝大多数的归结过程，所以我们可以允许耗费少量的准确性换取更快的推理速度，和更少的推理归结次数。运用长度限制后，运行结果如下。



我们可以看到，程序总共进行了 1148 次归结，相比于未使用长度限制策略的 1340 次，执行效率提升了 14.3%，并且得到最终结果“谋杀者是 A”。

综上，可以发现使用“长度限制”策略虽然会在一些极端情况下失去准确性，但在绝大多数情况下，使用该策略能一从程度缩短运行时间，提升运行效率，且不会过多地影响搜索的子句个数，最终归结结果正确。故可认为鉴于其特点，“长度限制”优化策略应当被使用。

2.4.3 包容归结策略

包容归结策略的核心就是要清楚所有被知识库中的已有语句包容（即，比该语句更特例）的语句。换言之，该法的核心就是要保证每一次新加入的子句都要包含原子句中所不具有的新信息——所谓新信息就是这条新语句不能由当前子句集中已有的条件推导为永真句。

以破案问题的子句举例，破案问题中存在如下子句：

C2: ~S(x1,A)|H(x1,A)

C3: ~H(A,x2)|~H(C,x2)

T1: ~S(A,A)

我们可以很容易发现 C2 与 C3 可以进行子句归结，归结结果为：

p: ~S(A,A)|~H(C,A)

在不使用包容归结策略的前提下，根据最基本的 BFS 策略，语句 p 所以应当进入子集库。但当我们仔细查看 p 子句时，不难发现因为 T1 为真，所以 p 的一个文字~S(A,A)也为真，所以 p 子句其实本质上是一个永真句。

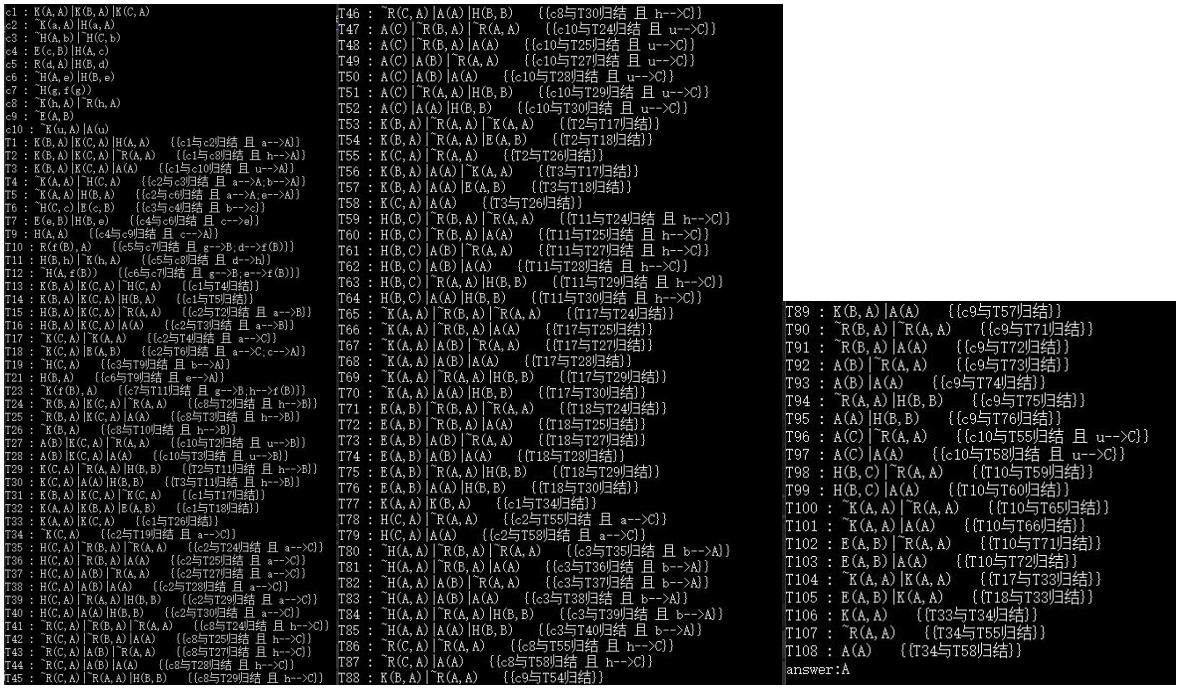
正是因为 T1 子句的存在，而且 T1 子句包容了 p 子句的内容，所以 p 子句的引入将不会再携带有任何新的信息，在这种情况下，就应该放弃 p 子句的引入，以使尽可能少的子句进入子集库，从而降低遍历次数。

该功能的实现就是将每一个试图进入子集库的语句进行先进行一次分析，将子集库中目前已

存的所有子句与 p 子句进行比较分析，如果库中存在一个子句的所有文字都在 p 中有与之相同的文字对应，那么 p 子句就被该语句包含，则不应该让该语句入库。

除此以外，值得一提的是，由于包容归结策略所拒绝入库的子句都是不携带新信息的子句，换言之，该策略没有拒绝任一携带新信息的子句入库，所以该策略是完备的，不会存在因为优化而导致归结推理出错的情况。

还是以破案问题为例，当我们仅使用长度限制优化策略执行程序时，需要进行 1148 次归结，但是当我们再加入包容规则后，运行结果如下。



根据以上运行结果可知，当采用包容归结策略时，程序仅需执行 108 次归结即可得出正确结果。相较于仅使用长度限制优化策略，效率提升达到 90.6%，极大地提升了程序执行效率。因此，我们可以看到包容规则的使用将极大地提升程序执行效率，避免了很多不必要的归结，并且由于包容规则是完备的，并不会因为效率的提升而丧失准确率。综上，包容规则在归结的应用上是极其有必要的。

尽管，程序仅需 108 次便可得到正确结果，但是由于程序仍然采用盲目的 bfs 搜索，仍然执行了很多无需搜索的步骤。在很多一眼便能看出的情况，仍然需要遍历很多多余的步骤，这显然将极大地降低执行效率。从最终回溯得到最优结果也可以看出还有很多值得优化的地方，使得程序更加智能。

3 实验过程

3.1 环境说明

操作系统：Window
开发语言：C++
编译平台：Visual Studio 2022
核心库 ：

```
<iostream>          // 标准流
<vector>            // SRL容器
<string>            // string
<queue>             // STL队列
<algorithm>         // 基础算法库
<iomanip>           // 格式化输入输出
<windows.h>         // windows系统库
<graphics.h>        // EasyX库
<conio.h>           // EasyX库
```

3.2 源代码文件清单

3.2.1 源代码清单

logicalReasoning_head.h：存储程序核心库声明、宏定义、结构体声明、函数声明等等程序，实现各个 cpp 文件之间的连接。

logicalReasoning_kernel.cpp：内部核心执行函数，用于实现一阶逻辑归结实验。

logicalReasoning_ui.cpp：实现一阶逻辑归结实验的 UI 界面。

logicalReasoning_main.cpp：存储一阶逻辑归结的主函数，实现总体功能的连接。

3.2.2 主要函数清单

Ⅰ. clauseResolution 类函数

```
// 内部函数
void initialRecord(void);
// 初始逻辑记录
void processRecord(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData, RECORD& get);
// 归结过程记录
void traceBackIn(RECORD& c);
// 寻找最佳归结路线内部递归函数
void expressionInput(string& original, CLAUSES& c);
// 将一个字符串转换成
bool clausesUpdate(CLAUSES& ans, vector<char>& record, vector<char>& upData);
// 子句更新
bool judgeAnswer(CLAUSES& c1);
// 判断是否归结到终点
bool equalDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 等式归结[若为E(f(A),B)则将f(A)替换为B]
bool containDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 包容归结
bool lengthDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 长度限制
bool twoClauseResolu(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& record);
```

```
// 两个子句归结
void processShow(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData);
// 控制台打印过程
void initialShow(void);
// 外部函数
void traceBack(void);
// 寻找最佳归结路线
void clausInput(vector<string> &originalSentence);
// 将输入子句转化为易计算结构子句
void resolution(void);
// 执行子句归结
void allShow(void);
// 展示所有归结过程
void bestShow(void);
// 展示最优归结过程
void ansShow(void);
// 显示最终结果
void tempShow(void);
// for test
```

II. clauseUI 类函数

```
clauseUI(void);
// 初始化
void bgShow(void);
// 展示背景
int orderChoose(bool haveIn = false);
// 按键指示
```

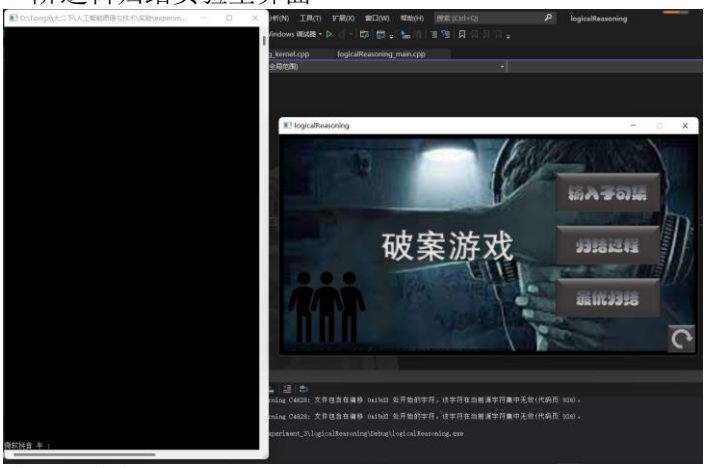
III. 其余工具函数

```
void dataIn(vector<string>& originalSentence);
//输入对应的数据
```

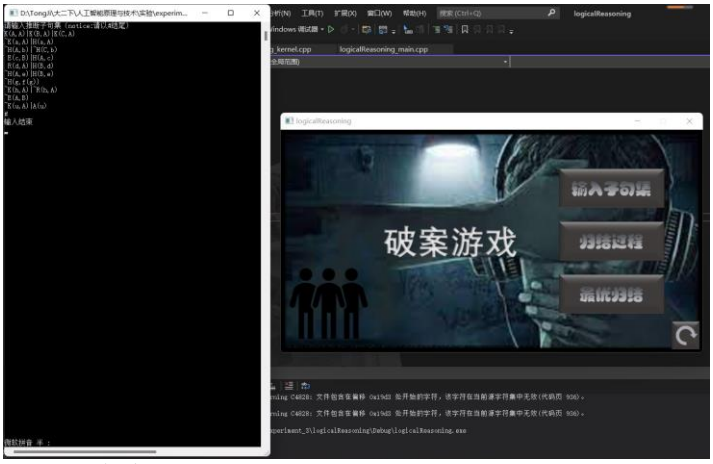
3.3 实验结果展示

3.3.1 界面结果展示

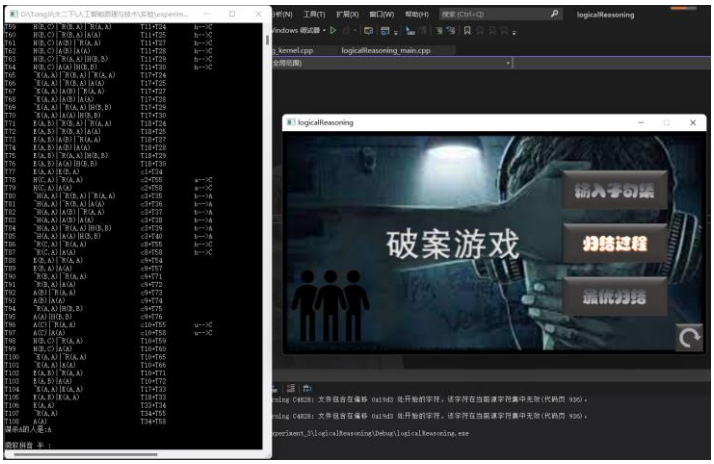
1. 一阶逻辑归结实验主界面



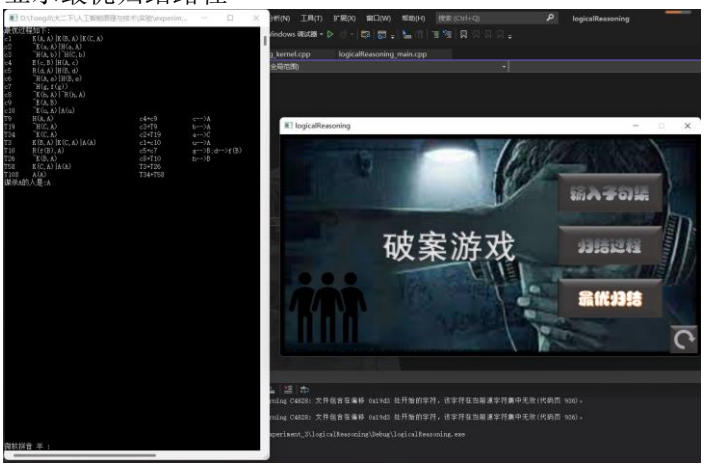
2. 输入子句集



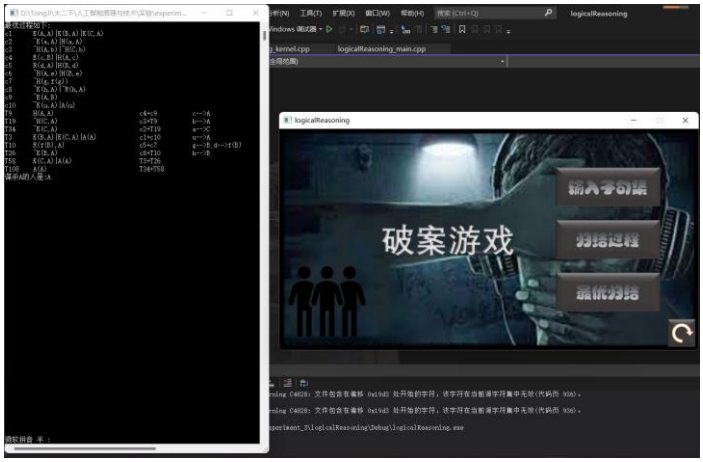
3. 显示归结过程



4. 显示最优归结路径



5. 重置



3.3.2 运行结果展示

首先考虑到本次实验的主要功能是实现子集编写的归结逻辑程序对破案问题进行推理，因此重点在于对归结过程的理解上，而非子句集的构造上。因此，实验采用人工的推导的方式生成子句集。

为了表述方便，给出谓词定义如下:

K(a,b):a 杀了 b

H(a,b):a 恨 b

E(a,b):a 和 b 是一样的　　//该函数也是本程序设定的 E 逻辑关键字

R(a,b):a 比 b 有钱

然后对题目给定条件使用以上谓词进行表示:

① K(A,A)∨K(B,A)∨K(C,A)

② ∀x，K(x,A)→H(x,A)

③ ∀x，H(A,x)→~H(C,x)

④ ∀x，~E(x,B)→H(A,x)

⑤ ∀x，~R(x,A)→H(B,x)

⑥ ∀x，H(A,x)→H(B,x)

⑦ ∀x ∃y，~H(x,y)

⑧ ∀x，K(x,A)→~R(x,A)

⑨ ~E(A,B)

紧接着，将这 9 个一阶逻辑子句依次经过消除蕴涵词、否定内移、变量标准化、Skolem 化、删除全称量词，将∧分配到∨中这几步操作后，再将结果转化成本程序可识别的子句输入，得到应当输入的内容为:（注:由于字符受限，采用|代替∨，其中 f(a)表示除 a 以外的所有人）

K(A,A)|K(B,A)|K(C,A)

~K(a,A)|H(a,A)

~H(A,b)|~H(C,b)

E(c,B)|H(A,c)

R(d,A)|H(B,d)

~H(A,e)|H(B,e)

~H(g,f(g))

~K(h,A)|~R(h,A)

~E(A,B)

最后需要对问题进行转化，将其用谓词公式进行表示。假定最终结果谓词为 A(u)，u 谋杀了 A。因此得到问题谓词 G:~K(u,A)|A(u)

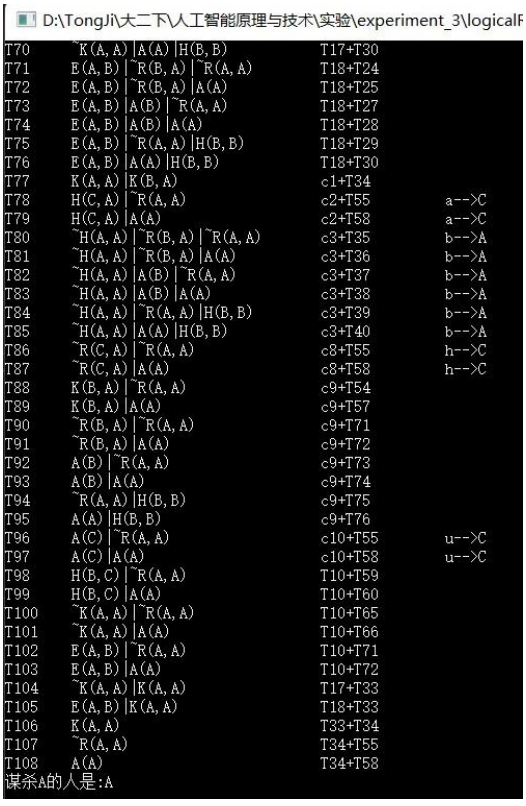
因此，得到最终输入子句集为：

K(A,A)|K(B,A)|K(C,A)
~K(a,A)|H(a,A)
~H(A,b)|~H(C,b)
E(c,B)|H(A,c)
R(d,A)|H(B,d)
~H(A,e)|H(B,e)
~H(g,f(g))
~K(h,A)|~R(h,A)
~E(A,B)
~K(u,A)|A(u)

将以上子句集输入至程序中，程序将进行循环归结，得到结果 A 为固定值时即为最终结果。程序执行如下:

1. **归结过程：**

此时加入长度限制、包容归结的优化策略后的全部归结结果。



归结过程如下:

c1	K(A,A)|K(B,A)|K(C,A)		
c2	~K(a,A)|H(a,A)		
c3	~H(A,b)|~H(C,b)		
c4	E(c,B)|H(A,c)		
c5	R(d,A)|H(B,d)		
c6	~H(A,e)|H(B,e)		
c7	~H(g,f(g))		
c8	~K(h,A)|~R(h,A)		
c9	~E(A,B)		
c10	~K(u,A)|A(u)		
T1	K(B,A)|K(C,A)|H(A,A)	c1+c2	a-->A
T2	K(B,A)|K(C,A)|~R(A,A)	c1+c8	h-->A
T3	K(B,A)|K(C,A)|A(A)	c1+c10	u-->A
T4	~K(A,A)|~H(C,A)	c2+c3	a-->A;b-->A
T5	~K(A,A)|H(B,A)	c2+c6	a-->A;e-->A
T6	~H(C,c)|E(c,B)	c3+c4	b-->c
T7	E(e,B)|H(B,e)	c4+c6	c-->e
T8	E(f(A),B)	c4+c7	g-->A;c-->f(A)
T9	H(A,A)	c4+c9	c-->A
T10	R(f(B),A)	c5+c7	g-->B;d-->f(B)
T11	H(B,h)|~K(h,A)	c5+c8	d-->h
T12	~H(A,f(B))	c6+c7	g-->B;e-->f(B)
T13	K(B,A)|K(C,A)|~H(C,A)	c1+T4	
T14	K(B,A)|K(C,A)|H(B,A)	c1+T5	
T15	H(B,A)|K(C,A)|~R(A,A)	c2+T2	a-->B
T16	H(B,A)|K(C,A)|A(A)	c2+T3	a-->B
T17	~K(C,A)|~K(A,A)	c2+T4	a-->C
T18	~K(C,A)|E(A,B)	c2+T6	a-->C;c-->A
T19	~H(C,A)	c3+T9	b-->A
T20	E(f(B),B)	c4+T12	c-->f(B)
T21	H(B,A)	c6+T9	e-->A
T22	E(f(B),B)	c7+T7	g-->B;e-->f(B)
T23	~K(f(B),A)	c7+T11	g-->B;h-->f(B)
T24	~R(B,A)|K(C,A)|~R(A,A)	c8+T2	h-->B
T25	~R(B,A)|K(C,A)|A(A)	c8+T3	h-->B

T26	~K(B,A)	c8+T10	h-->B
T27	A(B)|K(C,A)|~R(A,A)	c10+T2	u-->B
T28	A(B)|K(C,A)|A(A)	c10+T3	u-->B
T29	K(C,A)|~R(A,A)|H(B,B)	T2+T11	h-->B
T30	K(C,A)|A(A)|H(B,B)	T3+T11	h-->B
T31	K(B,A)|K(C,A)|~K(C,A)	c1+T17	
T32	K(A,A)|K(B,A)|E(A,B)	c1+T18	
T33	K(A,A)|K(C,A)	c1+T26	
T34	~K(C,A)	c2+T19	a-->C
T35	H(C,A)|~R(B,A)|~R(A,A)	c2+T24	a-->C
T36	H(C,A)|~R(B,A)|A(A)	c2+T25	a-->C
T37	H(C,A)|A(B)|~R(A,A)	c2+T27	a-->C
T38	H(C,A)|A(B)|A(A)	c2+T28	a-->C
T39	H(C,A)|~R(A,A)|H(B,B)	c2+T29	a-->C
T40	H(C,A)|A(A)|H(B,B)	c2+T30	a-->C
T41	~R(C,A)|~R(B,A)|~R(A,A)	c8+T24	h-->C
T42	~R(C,A)|~R(B,A)|A(A)	c8+T25	h-->C
T43	~R(C,A)|A(B)|~R(A,A)	c8+T27	h-->C
T44	~R(C,A)|A(B)|A(A)	c8+T28	h-->C
T45	~R(C,A)|~R(A,A)|H(B,B)	c8+T29	h-->C
T46	~R(C,A)|A(A)|H(B,B)	c8+T30	h-->C
T47	A(C)|~R(B,A)|~R(A,A)	c10+T24	u-->C
T48	A(C)|~R(B,A)|A(A)	c10+T25	u-->C
T49	A(C)|A(B)|~R(A,A)	c10+T27	u-->C
T50	A(C)|A(B)|A(A)	c10+T28	u-->C
T51	A(C)|~R(A,A)|H(B,B)	c10+T29	u-->C
T52	A(C)|A(A)|H(B,B)	c10+T30	u-->C
T53	K(B,A)|~R(A,A)|~K(A,A)	T2+T17	
T54	K(B,A)|~R(A,A)|E(A,B)	T2+T18	
T55	K(C,A)|~R(A,A)	T2+T26	
T56	K(B,A)|A(A)|~K(A,A)	T3+T17	
T57	K(B,A)|A(A)|E(A,B)	T3+T18	
T58	K(C,A)|A(A)	T3+T26	
T59	H(B,C)|~R(B,A)|~R(A,A)	T11+T24	h-->C
T60	H(B,C)|~R(B,A)|A(A)	T11+T25	h-->C
T61	H(B,C)|A(B)|~R(A,A)	T11+T27	h-->C

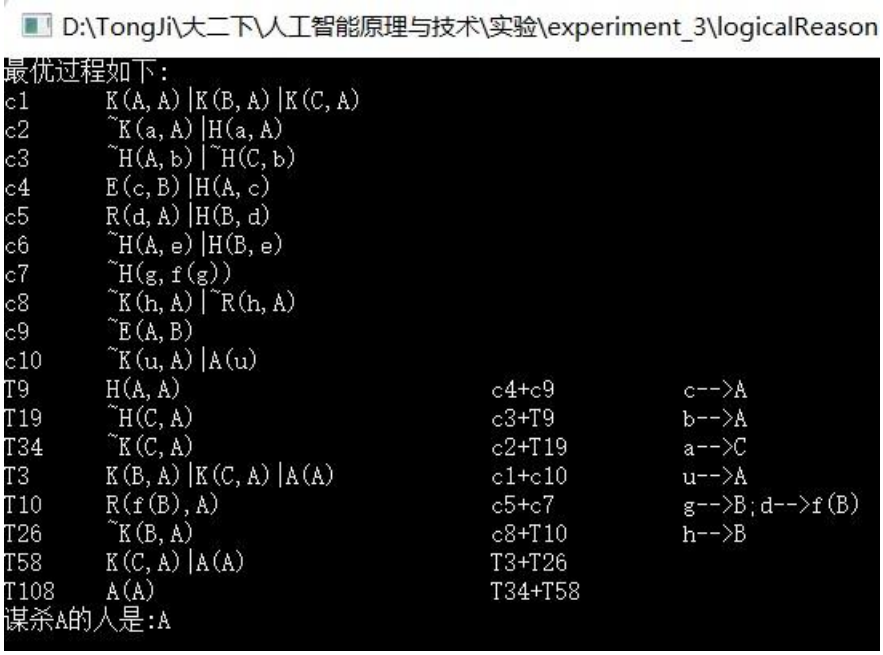
T62	H(B,C)|A(B)|A(A)	T11+T28	h-->C
T63	H(B,C)|~R(A,A)|H(B,B)	T11+T29	h-->C
T64	H(B,C)|A(A)|H(B,B)	T11+T30	h-->C
T65	~K(A,A)|~R(B,A)|~R(A,A)	T17+T24	
T66	~K(A,A)|~R(B,A)|A(A)	T17+T25	
T67	~K(A,A)|A(B)|~R(A,A)	T17+T27	
T68	~K(A,A)|A(B)|A(A)	T17+T28	
T69	~K(A,A)|~R(A,A)|H(B,B)	T17+T29	
T70	~K(A,A)|A(A)|H(B,B)	T17+T30	
T71	E(A,B)|~R(B,A)|~R(A,A)	T18+T24	
T72	E(A,B)|~R(B,A)|A(A)	T18+T25	
T73	E(A,B)|A(B)|~R(A,A)	T18+T27	
T74	E(A,B)|A(B)|A(A)	T18+T28	
T75	E(A,B)|~R(A,A)|H(B,B)	T18+T29	
T76	E(A,B)|A(A)|H(B,B)	T18+T30	
T77	K(A,A)|K(B,A)	c1+T34	
T78	H(C,A)|~R(A,A)	c2+T55	a-->C
T79	H(C,A)|A(A)	c2+T58	a-->C
T80	~H(A,A)|~R(B,A)|~R(A,A)	c3+T35	b-->A
T81	~H(A,A)|~R(B,A)|A(A)	c3+T36	b-->A
T82	~H(A,A)|A(B)|~R(A,A)	c3+T37	b-->A
T83	~H(A,A)|A(B)|A(A)	c3+T38	b-->A
T84	~H(A,A)|~R(A,A)|H(B,B)	c3+T39	b-->A
T85	~H(A,A)|A(A)|H(B,B)	c3+T40	b-->A
T86	~R(C,A)|~R(A,A)	c8+T55	h-->C
T87	~R(C,A)|A(A)	c8+T58	h-->C
T88	K(B,A)|~R(A,A)	c9+T54	
T89	K(B,A)|A(A)	c9+T57	
T90	~R(B,A)|~R(A,A)	c9+T71	
T91	~R(B,A)|A(A)	c9+T72	
T92	A(B)|~R(A,A)	c9+T73	
T93	A(B)|A(A)	c9+T74	
T94	~R(A,A)|H(B,B)	c9+T75	
T95	A(A)|H(B,B)	c9+T76	
T96	A(C)|~R(A,A)	c10+T55	u-->C
T97	A(C)|A(A)	c10+T58	u-->C

T98	H(B,C)|~R(A,A)	T10+T59
T99	H(B,C)|A(A)	T10+T60
T100	~K(A,A)|~R(A,A)	T10+T65
T101	~K(A,A)|A(A)	T10+T66
T102	E(A,B)|~R(A,A)	T10+T71
T103	E(A,B)|A(A)	T10+T72
T104	~K(A,A)|K(A,A)	T17+T33
T105	E(A,B)|K(A,A)	T18+T33
T106	K(A,A)	T33+T34
T107	~R(A,A)	T34+T55
T108	A(A)	T34+T58

谋杀 A 的人是:A

2. 最优归结路径：

由于仍然采用 bfs 搜索，存在较多的重复步骤，因此，我们可以根据最终结果回溯得到一条最优归结路径，避免了很多不必要的归结过程。可以看出，真正的有用归结只需要 8 步而已。



最优过程如下:	
c1	K(A,A)|K(B,A)|K(C,A)
c2	~K(a,A)|H(a,A)
c3	~H(A,b)|~H(C,b)
c4	E(c,B)|H(A,c)

c5	R(d,A)|H(B,d)		
c6	~H(A,e)|H(B,e)		
c7	~H(g,f(g))		
c8	~K(h,A)|~R(h,A)		
c9	~E(A,B)		
c10	~K(u,A)|A(u)		
T9	H(A,A)	c4+c9	c-->A
T19	~H(C,A)	c3+T9	b-->A
T34	~K(C,A)	c2+T19	a-->C
T3	K(B,A)|K(C,A)|A(A)	c1+c10	u-->A
T10	R(f(B),A)	c5+c7	g-->B;d-->f(B)
// 过程中执行 E(f(B),B)的等价归结			
T26	~K(B,A)	c8+T10	h-->B
T58	K(C,A)|A(A)	T3+T26	
T108	A(A)	T34+T58	
谋杀 A 的人是:A			

3.4 实验结论

根据实验结果可以看出，本次一阶归结实验的实现效果较好，程序合理的进行归结推理，得出谋杀者为 A 的结论。过程中使用较多的优化策略，例如长度限制以及包容归结策略等，方法都极大地提升了程序运行效率，在保证准确推理的基础上，获得了使用更少的时间。从以上具体归结步骤，可以看出程序的正确性。

总体而言，本次实验基本达到预期结果，成果较好。

4 总结

4.1 实验中存在的问题及解决方案

在实验过程中也遇到较多困难，其中花费较长时间的是对 E 逻辑的处理上，最开始没有意识到等价的逻辑内涵，导致一直无法运行成功。后续经过手动完成推理过程，发现等价逻辑内涵在归结过程中的应用，才意识到需要在归结过程中对等价的二者进行统一，这样才能推导出最终结果。在采用 E 等价逻辑后，便能顺利得到正确结果。

4.2 心得体会

经过本次实验，我对于一阶逻辑、归结原理以及置换合一方法有了更深的了解，尤其是对归结原理的具体使用上。使用归结规则可以基于现有的条件推导出正确的结论，起初对于这样的方式感到惊奇，但后续自行推导实现求得最终结论后，也觉得理所当然。通过对其中步骤的亲自实现，对归结内部的原理和细节有了更深的了解，这也激励我之后在这方面进一步学习，做到知其然还要知其所以然。

4.3 后续改进方向

后续关注如何归结原理的程序实现上进一步优化，找到更优秀的归结方式，能迅速找到对得到最终结果最有效的归结步骤，缩短不必要的归结耗费，提升程序的智能。

与此同时，也可以了解更多基于逻辑的推理方法，学习人工智能领域更多基础的知识，进一步夯实基础。

4.4 总结

通过本次实验，让我进一步感叹逻辑的强大，仅仅通过现有的知识，加上逻辑推理便可得到答案。总体而言，本次实验完成了既定目标，并且程序执行的优化上试图寻找更多的方案，极大地提升了程序运行效率。因此，综上：本次实验完成度较高，关注了更多的知识，并且个人也收获颇丰，激励我在人工智能领域进一步学习提升。

参考文献

[1]王湘云.一阶谓词逻辑在人工智能知识表示中的应用[J].重庆工学院学报(社会科学版),2007(09):69-71.

[2]徐从富,郝春亮,苏保君,楼俊杰.马尔可夫逻辑网络研究[J].软件学报,2011,22(08):1699-1713.

[3]邹丽. 基于语言真值格蕴涵代数的格值命题逻辑及其归结自动推理研究[D].西南交通大学,2010.

[4]刘叙华.Horn 集上的输入半锁归结原理[J].科学通报,1985(16):1201-1202.

附录

logicalReasoning_head.h

```
/*
 * @author   : gonzalez
 * @time     : 2022.5.26-2022.6.2
 * @function : 项目头文件
 */
#pragma once

#include<iostream>              // 标准流
#include<vector>                // SRL容器
#include<string>                // string
#include<queue>                 // STL队列
#include<algorithm>             // 基础算法库
#include<iomanip>               // 格式化输入输出
#include<windows.h>             // windows系统库
#include<graphics.h>            // EasyX库
#include<conio.h>               // EasyX库

using namespace std;

//*************************************************************************
//* @function  : 一阶逻辑内部核心实现
//* @notice    : 无
//*************************************************************************

// @notice:
// K(x,y)  表示  怀疑x杀了y
// L(x)    表示  x生活在这栋房子里
// H(x,y)  表示  x恨y
// R(x,y)  表示  x比y富有
// E(x,y)  表示  x和y相等
// A(x)    表示  x就是所求凶手
//
// 变量 -- a-z
// 常量 -- A-Z
// 函数 -- f(x)用'&'表示函数

// @function : 长度归结限制长度
#define  LENGTHEDGE      3
// @function : 函数代表
#define  FUNCTION       '&'
// @function : 表达式划归
#define  OR             '|'
#define  NOT            '~'
#define  DOT            ','
#define  LEFTBRACKET    '('
#define  RIGHTBRACKET   ')'
#define  EQUAL          'E'
#define  ANSWER         'A'
```

```
// 函数结构体
struct FUNC {
    char name = '\0';
    char parameter = '\0';
};
// 独立表达式
struct EXPRE{
    bool IsNot = true;        // 表达式是否取非
    char type = '\0';         // 表达式的类别
    vector<char>element;      // 表达式中的元素
    FUNC func;                // 单独存储函数表达式
};
// 独立子句
struct CLAUSES{
    string name;              // 子句编号
    vector<EXPRE>expressions; // 子句表达式合集
};
// 过程记录
struct RECORD{
    string resultName;   // 结果逻辑名字
    string result;       // 逻辑表达式
    string father1Name;  // 父节点1名字
    string father2Name;  // 父节点2名字
    string change;       // 置换合一变化
};

// @function : 子句归结执行内核
class clauseResolution {
protected:

    // 内部参数
    CLAUSES answer;                                                    // 最终结果
    int countNum = 0;                                                  // 用于记录转换个数
    vector<char>expressionType;                                        // 该推断问题中表达式的
所有种类 --> 'A'-'Z'代替
    vector<CLAUSES>clauseBag;                                          // 存储所有转化后的子句
集

    // 可视化结果存储
    vector<RECORD>initialList;                                         // 初始化数据存储
    vector<RECORD>recordList;                                          // 过程数据存储
    vector<RECORD>bestPath;                                            // 最优归结路线

    // 内部函数
    void initialRecord(void);
// 初始逻辑记录
    void processRecord(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData, RECORD&
get); // 归结过程记录
    void traceBackIn(RECORD& c);
// 寻找最佳归结路线内部递归函数
    void expressionInput(string& original, CLAUSES& c);
// 将一个字符串转换成
```

```
    bool clausesUpdate(CLAUSES& ans, vector<char>& record, vector<char>& upData);
// 子句更新
    bool judgeAnswer(CLAUSES& c1);
// 判断是否归结到终点
    bool equalDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 等式归结[若为E(f(A),B)则将f(A)替换为B]
    bool containDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 包容归结
    bool lengthDeal(CLAUSES& c, vector<CLAUSES>& tempBag);
// 长度限制
    bool twoClauseResolu(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& record);
// 两个子句归结
    void processShow(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData);
// 控制台打印过程
    void initialShow(void);
// 控制台打印初始过程


public:
    // 外部函数
    void traceBack(void);                                                   // 寻找最佳归结路线
    void clausInput(vector<string> &originalSentence);                      // 将输入子句转化为易计
算结构子句
    void resolution(void);                                                  // 执行子句归结
    void allShow(void);                                                     // 展示所有归结过程
    void bestShow(void);                                                    // 展示最优归结过程
    void ansShow(void);                                                     // 显示最终结果

    void tempShow(void);                                                    // for test
};



//*********************************************************************
//* @function  : UI界面类
//* @notice    : 无
//*********************************************************************

// 页面布局结构体
struct SURFACE {
    short left = 0;
    short top = 0;
};

#define UILENGTH   758
#define UIWIDTH    400

#define BUTTONLENGTH 189
#define BUTTONWIDTH  70
#define RESTARTLEN   50


#define MOUSEINPUT    1
#define MOUSEALL      2
```

```
#define MOUSEBEST     3
#define MOUSEREST     4

// @function : 字句归结UI界面
class clauseUI {
protected:
     // 界面布局结构体定义
     SURFACE backArea;
     SURFACE inputArea;
     SURFACE showallArea;
     SURFACE showbestArea;
     SURFACE restartArea;

     // 界面图片调用定义
     IMAGE background;
     IMAGE input;
     IMAGE inputActive;
     IMAGE showAll;
     IMAGE showAllActive;
     IMAGE showBest;
     IMAGE showBestActive;
     IMAGE restart;
     IMAGE restartActive;

public:
     clauseUI(void);                              // 初始化
     void bgShow(void);                           // 展示背景
     int orderChoose(bool haveIn = false);     // 按键指示

};



//*************************************************************************
//* @function  : 工具函数
//* @notice    :
//*************************************************************************

// @function : 工具函数
void dataIn(vector<string>& originalSentence);//输入对应的数据
```

logicalReasoning_kernel.cpp

```
/*
 * @author   : gonzalez
 * @time     : 2022.5.26-2022.6.2
 * @function :
 */

#include"logicalReasoning_head.h"
using namespace std;
```

```
/*********************************************************************************************/
/*****************************一阶逻辑归结内部核心函数*****************************************/
/*********************************************************************************************/

// @function : 表达式子句划归
void clauseResolution::expressionInput(string& original, CLAUSES& c) {

    // 表达式数据
    EXPRE expression;

    // 判断类型
    string getString = original;
    int st = 0;
    if (getString[0] == NOT) {
        expression.IsNot = false;
        st = 1;
    }
    else
        expression.IsNot = true;

    // 表达式添加新类型
    bool hadFlag = false;
    for (int i = 0; i < (int)expressionType.size(); i++)
        if (expressionType[i] == getString[st]) {
            hadFlag = true;
            break;
        }
    if (!hadFlag)
        expressionType.push_back(getString[st]);
    expression.type = getString[st];

    // 此后仅考虑'(exprenssion)'
    ++st;
    getString = getString.substr(st);

    // 判断内部数据
    int num = count(getString.begin(), getString.end(), DOT);
    // 表达式内两个字符
    if (num != 0) {

        // 初始化
        int site = getString.find(DOT);
        string frontString = getString;
        frontString.erase(site);
        string backString = getString.substr(site + 1);

        // 判断前半部
        if (count(frontString.begin(), frontString.end(), RIGHTBRACKET)) {     // 函数特判
            expression.element.push_back(FUNCTION);
            expression.func.name = *(frontString.begin() + 1);
            expression.func.parameter = *(frontString.begin() + 1 + 1 + 1);
        }
        else {
```

```
            char front = frontString[frontString.size() - 1];
            expression.element.push_back(front);
        }

        // 判断后半部
        if (count(backString.begin(), backString.end(), LEFTBRACKET)) {        // 函数特判
            expression.element.push_back(FUNCTION);
            expression.func.name = *(backString.begin());
            expression.func.parameter = *(backString.begin() + 1 + 1);
        }
        else {
            char back = backString[0];
            expression.element.push_back(back);
        }


    }
    // 表达式内仅一个字符
    else {
        // 判断是否为函数
        if (count(getString.begin(), getString.end(), RIGHTBRACKET) == 2) {    // 函数特判
            expression.element.push_back(FUNCTION);
            expression.func.name = *(getString.begin() + 1);
            expression.func.parameter = *(getString.begin() + 1 + 1 + 1);
        }
        // 判断常量或变量
        else {
            char val = getString[1];
            expression.element.push_back(val);
        }
    }

    // 添加对应子句
    c.expressions.push_back(expression);

    return;
}

// @function : 读入字符串转机内子句
void clauseResolution::clausInput(vector<string>& originalSentence){

    int len = originalSentence.size();
    for (int i = 0; i < len; i++) {
        CLAUSES c;                                  // 子句获取
        string temp = originalSentence[i];
        int num = count(temp.begin(), temp.end(), OR);
        for (int i = 0; i <= num; i++) {
            if (i == num)
                expressionInput(temp,c);
            else {
                int site = temp.find(OR);
                string subTemp = temp;
                subTemp.erase(site);
                temp = temp.substr(site + 1);     // 删去对应的'|'
```

```
                    expressionInput(subTemp,c);
                }
            }
            c.name = "c" + to_string(i + 1);          // 子句名字赋值
            clauseBag.push_back(c);                   // 添加子句
        }
        return;
}

// @function : 两个子句进行归结
bool clauseResolution::twoClauseResolu(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>&
record) {

        bool flag = false;
        for (int c1i = 0; c1i < (int)c1.expressions.size(); c1i++) {
            for (int c2j = 0; c2j < (int)c2.expressions.size(); c2j++) {
                flag = true;
                if (c1.expressions[c1i].type == c2.expressions[c2j].type)
{                                   // 表达式类型一致
                    if (c1.expressions[c1i].IsNot != c2.expressions[c2j].IsNot)
{                               // 表达式符号相反
                        if (c1.expressions[c1i].element.size() ==
c2.expressions[c2j].element.size()) {       // 参数个数一致
                            vector<char> recordTemp;
// 暂时进行归结存储
                            // 判断具体各个是否能合一置换
                            for (int k = 0; k < (int)c1.expressions[c1i].element.size(); k++) {
                                //// 为存在函数的情况
                                //if(c1.expressions[c1i].element[k]==FUNCTION)
                                //  if (c1.expressions[c1i].func.parameter >= 'A' &&
c1.expressions[c1i].func.parameter <= 'Z')
                                //      if (c2.expressions[c2j].element[k] >= 'A' &&
c2.expressions[c2j].element[k] <= 'Z') {
                                //          flag = false;
// 不可进行归结
                                //          break;
                                //      }
                                //if (c2.expressions[c1i].element[k] == FUNCTION)
                                //  if (c2.expressions[c1i].func.parameter >= 'A' &&
c2.expressions[c1i].func.parameter <= 'Z')
                                //      if (c1.expressions[c2j].element[k] >= 'A' &&
c1.expressions[c2j].element[k] <= 'Z') {
                                //          flag = false;
// 不可进行归结
                                //          break;
                                //      }
                                //// 均为常量
                                //if ((c1.expressions[c1i].element[k] >= 'A' &&
c1.expressions[c1i].element[k] <= 'Z') &&
                                //  (c2.expressions[c2j].element[k] >= 'A' &&
c2.expressions[c2j].element[k] <= 'Z')) {
                                //  if (c1.expressions[c1i].element[k] !=
c2.expressions[c2j].element[k]) {
                                //      flag = false;
```

```
// 不可进行归结
                                //        break;
                                //    }
                                //}

                                // 至少一个变量||均常量且相等  -->  可归结
                                if ((c1.expressions[c1i].element[k] >= 'a' &&
c1.expressions[c1i].element[k] < 'z') ||
                                    (c2.expressions[c2j].element[k] >= 'a' &&
c2.expressions[c2j].element[k] < 'z') ||
                                    (c1.expressions[c1i].element[k] ==
c2.expressions[c2j].element[k]))
                                    flag = true;
                                else {
                                    flag = false;
                                    break;
                                }
                                recordTemp.push_back(c1.expressions[c1i].element[k]);// 归结
过程统计
                                if (c1.expressions[c1i].element[k] == FUNCTION) {
                                    recordTemp.push_back(c1.expressions[c1i].func.name);

    recordTemp.push_back(c1.expressions[c1i].func.parameter);
                                }
                                recordTemp.push_back(c2.expressions[c2j].element[k]);// 归结
过程统计
                                if (c2.expressions[c2j].element[k] == FUNCTION) {
                                    recordTemp.push_back(c2.expressions[c2j].func.name);

    recordTemp.push_back(c2.expressions[c2j].func.parameter);
                                }
                            }
                            // 进行归结
                            if (flag) {
                                // 新子句名字
                                ans.name = "T" + to_string(++countNum);

                                // 添加原有子句
                                for (int tempi = 0; tempi < (int)c1.expressions.size(); tempi++)
{
                                    if (tempi != c1i)
                                        ans.expressions.push_back(c1.expressions[tempi]);
                                }
                                // 添加原有子句
                                for (int tempj = 0; tempj < (int)c2.expressions.size(); tempj++)
{
                                    if (tempj != c2j)
                                        ans.expressions.push_back(c2.expressions[tempj]);
                                }
                                // 合一置换等式语句
                                for (int i = 0; i < (int)recordTemp.size(); i++)
                                    record.push_back(recordTemp[i]);

                                return true;
```

```
                            }
                        }
                    }
                }
            }

    return false;

}

// @function : 新子句标识符更新
bool clauseResolution::clausesUpdate(CLAUSES& ans, vector<char>& record, vector<char>& upData)
{

    for (int i = 0; i < (int)record.size();) {

        // 交换参数获取
        char get1[3] = { '\0','\0' ,'\0'};
        char get2[3] = { '\0','\0' ,'\0' };
        if (record[i] == FUNCTION) {
            get1[0] = FUNCTION;
            get1[1] = record[i + 1];
            get1[2] = record[i + 2];
            i += 3;
        }
        else {
            get1[0] = record[i];
            i += 1;
        }
        if (record[i] == FUNCTION) {
            get2[0] = FUNCTION;
            get2[1] = record[i + 1];
            get2[2] = record[i + 2];
            i += 3;
        }
        else {
            get2[0] = record[i];
            i += 1;
        }

        // 此时存在替换情况-->即1常量/1变量/1函数 代替 1变量
        if ((get1[0] >= 'a' && get1[0] <= 'z') || (get2[0] >= 'a' && get2[0] <= 'z')) {

            // 确定替换项和被替换项
            char cans, para;
            if ((get1[0] >= 'a' && get1[0] <= 'z')) {
                para = get1[0];
                cans = get2[0];
            }
            else {
                para = get2[0];
                cans = get1[0];
            }
```

```
            // 变化矩阵同步更新
            for (int ini = 0; ini < (int)upData.size(); ini++)
                if (upData[ini] == para)
                    upData[ini] = cans;

            // upData参数追加
            upData.push_back(para);//被替换变量
            upData.push_back(cans);//被替换常量
            // 函数参数追加
            FUNC temp = { '\0','\0' };
            if (get1[0] == FUNCTION) {
                upData.push_back(get1[1]);//函数符号
                temp.name = get1[1];
                upData.push_back(get1[2]);//函数参数
                temp.parameter = get1[2];
            }
            else if (get2[0] == FUNCTION) {
                upData.push_back(get2[1]);//函数符号
                temp.name = get2[1];
                upData.push_back(get2[2]);//函数参数
                temp.parameter = get2[2];
            }

            // 开始更新当前新子句信息
            for (int i = 0; i < (int)ans.expressions.size(); i++) {
                for (int j = 0; j < (int)ans.expressions[i].element.size(); j++) {
                    // 参数更换
                    if (para == ans.expressions[i].element[j]) {
                        ans.expressions[i].element[j] = cans;
                        ans.expressions[i].func.name = temp.name;
                        ans.expressions[i].func.parameter = temp.parameter;
                    }
                    // 函数参数更换
                    if (para == ans.expressions[i].func.parameter)
                        ans.expressions[i].func.parameter = cans;
                }
            }

            // 变化矩阵同步更新
            for (int ini = i; ini < (int)record.size(); ini++)
                if (record[ini] == para)
                    record[ini] = cans;
        }
    }
    return true;

}

// @function : 判断是否归结到终止符
bool clauseResolution::judgeAnswer(CLAUSES& c1) {
    if (c1.expressions.size() == 1)
        if (c1.expressions[0].IsNot == true)
            if (c1.expressions[0].type == ANSWER)
```

```
                if (c1.expressions[0].element.size() == 1)
                    if (c1.expressions[0].element[0] >= 'A' && c1.expressions[0].element[0]
<= 'Z')
                        return true;
    return false;
}

// ****************************
// * @NOTICE   : 优化改进策略  *
// ****************************

// @function : EQUAL等式归结
bool clauseResolution::equalDeal(CLAUSES& c,vector<CLAUSES>& tempBag) {
    if (c.expressions.size() == 1) {
        if (c.expressions[0].IsNot == true) {
            if (c.expressions[0].type == EQUAL) {
                char ori = c.expressions[0].element[0];
                char sub = c.expressions[0].element[1];
                FUNC func;
                func.name = c.expressions[0].func.name;
                func.parameter = c.expressions[0].func.parameter;
                // 开始全局替换
                if (((ori >= 'A' && ori <= 'Z') || (ori == FUNCTION && func.parameter >= 'A'
&& func.parameter <= 'Z'))
                    && ((sub > 'A' && sub <= 'Z') || (ori == FUNCTION && func.parameter >= 'A'
&& func.parameter <= 'Z'))) {

                    // clauseBag数据替换
                    for (int i = 0; i < (int)clauseBag.size(); i++) {
                        for (int j = 0; j < (int)clauseBag[i].expressions.size(); j++) {
                            for (int k = 0; k <
(int)clauseBag[i].expressions[j].element.size(); k++) {
                                if (clauseBag[i].expressions[j].element[k] == ori) {
                                    if (ori == FUNCTION) {
                                        if (clauseBag[i].expressions[j].func.parameter
== func.parameter) {
                                            clauseBag[i].expressions[j].element[k] =
sub;
                                            clauseBag[i].expressions[j].func.name =
'\0';
                                            clauseBag[i].expressions[j].func.parameter
= '\0';
                                        }
                                    }
                                    else if (sub == FUNCTION) {
                                        clauseBag[i].expressions[j].element[k] = sub;
                                        clauseBag[i].expressions[j].func.name =
func.name;
                                        clauseBag[i].expressions[j].func.parameter =
func.parameter;
                                    }
                                    else {
                                        clauseBag[i].expressions[j].element[k] = sub;
                                    }
```

```
                                    }
                                }
                            }
                        }

                        // tempBag数据替换
                        for (int i = 0; i < (int)tempBag.size(); i++) {
                            for (int j = 0; j < (int)tempBag[i].expressions.size(); j++) {
                                for (int k = 0; k <
(int)tempBag[i].expressions[j].element.size(); k++) {
                                    if (tempBag[i].expressions[j].element[k] == ori) {
                                        if (ori == FUNCTION) {
                                            if (tempBag[i].expressions[j].func.parameter ==
func.parameter) {
                                                tempBag[i].expressions[j].element[k] = sub;
                                                tempBag[i].expressions[j].func.name = '\0';
                                                tempBag[i].expressions[j].func.parameter =
'\0';
                                            }
                                        }
                                        else if (sub == FUNCTION) {
                                            tempBag[i].expressions[j].element[k] = sub;
                                            tempBag[i].expressions[j].func.name = func.name;
                                            tempBag[i].expressions[j].func.parameter =
func.parameter;
                                        }
                                        else {
                                            tempBag[i].expressions[j].element[k] = sub;
                                        }
                                    }
                                }
                            }
                        }

                        return true;
                    }
                }
            }
        }

        return false;

    }

    // @function : 包容归结函数
    bool clauseResolution::containDeal(CLAUSES& c, vector<CLAUSES>& tempBag) {

        // 输入长度初始化
        int len = c.expressions.size();

        // 原clauseBag中是否存在包含子句
        for (int outSite = 0; outSite < (int)clauseBag.size(); outSite++) {
            if ((int)clauseBag[outSite].expressions.size() <= len) {
                vector<bool> flag((int)clauseBag[outSite].expressions.size(), false);
```

```
            for (int each = 0; each < (int)clauseBag[outSite].expressions.size(); each++) {
                for (int inSite = 0; inSite < (int)c.expressions.size(); inSite++) {
                    bool flagIn = true;
                    if (clauseBag[outSite].expressions[each].IsNot ==
c.expressions[inSite].IsNot)
                        if (clauseBag[outSite].expressions[each].type ==
c.expressions[inSite].type)
                            if (clauseBag[outSite].expressions[each].element.size() ==
c.expressions[inSite].element.size())
                                for (int k = 0; k <
(int)clauseBag[outSite].expressions[each].element.size(); k++) {
                                    // 均为变量
                                    if
(clauseBag[outSite].expressions[each].element[k] >= 'A' &&
clauseBag[outSite].expressions[each].element[k] <= 'Z')
                                        if
(clauseBag[outSite].expressions[each].element[k] != c.expressions[inSite].element[k]) {
                                            flagIn = false;
                                            break;
                                        }
                                    // 均为常量
                                    if
(clauseBag[outSite].expressions[each].element[k] >= 'a' &&
clauseBag[outSite].expressions[each].element[k] <= 'z')
                                        if (!(c.expressions[inSite].element[k] >= 'a' &&
c.expressions[inSite].element[k] <= 'z')) {
                                            flagIn = false;
                                            break;
                                        }
                                    // 均为函数
                                    if (clauseBag[outSite].expressions[each].element[k]
== FUNCTION)
                                        if (c.expressions[inSite].element[k] != FUNCTION)
{
                                            flagIn = false;
                                            break;
                                        }
                                        else if
(clauseBag[outSite].expressions[each].func.parameter >= 'A' &&
clauseBag[outSite].expressions[each].func.parameter >= 'Z') {
                                            if (c.expressions[inSite].func.parameter !=
clauseBag[outSite].expressions[each].func.parameter) {
                                                flagIn = false;
                                                break;
                                            }
                                        }
                                        else if
(clauseBag[outSite].expressions[each].func.parameter >= 'a' &&
clauseBag[outSite].expressions[each].func.parameter >= 'z') {
                                            if
(!(c.expressions[inSite].func.parameter >='a' && c.expressions[inSite].func.parameter <='z'))
{
                                                flagIn = false;
                                                break;
```

```
                                        }
                                    }
                                }
                            else
                                flagIn = false;
                        else
                            flagIn = false;
                    else
                        flagIn = false;

                    // 判断是否存在一致的表达式
                    if (flagIn) {
                        flag[each] = flagIn;
                        break;
                    }
                }
            }
            bool getAns = true;
            for (int i = 0; i < (int)flag.size(); i++)
                if (!flag[i]) {
                    getAns = false;
                    break;
                }
            if (getAns)
                return true;// 全部存在一致的表达式
        }
    }

    // tempBag中是否存在包含子句
    for (int outSite = 0; outSite < (int)tempBag.size(); outSite++) {
        if ((int)tempBag[outSite].expressions.size() <= len) {
            vector<bool> flag((int)tempBag[outSite].expressions.size(), false);
            for (int each = 0; each < (int)tempBag[outSite].expressions.size(); each++) {
                for (int inSite = 0; inSite < (int)c.expressions.size(); inSite++) {
                    bool flagIn = true;
                    if (tempBag[outSite].expressions[each].IsNot ==
c.expressions[inSite].IsNot)
                        if (tempBag[outSite].expressions[each].type ==
c.expressions[inSite].type)
                            if (tempBag[outSite].expressions[each].element.size() ==
c.expressions[inSite].element.size())
                                for (int k = 0; k <
(int)tempBag[outSite].expressions[each].element.size(); k++) {
                                    // 均为常量
                                    if (tempBag[outSite].expressions[each].element[k] >=
'A' && tempBag[outSite].expressions[each].element[k] <= 'Z')
                                        if
(tempBag[outSite].expressions[each].element[k] != c.expressions[inSite].element[k]) {
                                            flagIn = false;
                                            break;
                                        }
                                    // 均为变量
                                    if (tempBag[outSite].expressions[each].element[k] >=
'a' && tempBag[outSite].expressions[each].element[k] <= 'z')
```

```
                                        if (!(c.expressions[inSite].element[k] >= 'a' &&
c.expressions[inSite].element[k] <= 'z')) {
                                            flagIn = false;
                                            break;
                                        }
                                    // 均为函数
                                    if (tempBag[outSite].expressions[each].element[k] ==
FUNCTION)
                                        if (c.expressions[inSite].element[k] != FUNCTION)
{
                                            flagIn = false;
                                            break;
                                        }
                                        else if
(tempBag[outSite].expressions[each].func.parameter >= 'A' &&
tempBag[outSite].expressions[each].func.parameter >= 'Z') {
                                            if (c.expressions[inSite].func.parameter !=
tempBag[outSite].expressions[each].func.parameter) {
                                                flagIn = false;
                                                break;
                                            }
                                        }
                                        else if
(tempBag[outSite].expressions[each].func.parameter >= 'a' &&
tempBag[outSite].expressions[each].func.parameter >= 'z') {
                                            if
(!(c.expressions[inSite].func.parameter >='a' && c.expressions[inSite].func.parameter <='z'))
{
                                                flagIn = false;
                                                break;
                                            }
                                        }
                                    }
                                else
                                    flagIn = false;
                            else
                                flagIn = false;
                        else
                            flagIn = false;

                        // 判断是否存在一致的表达式
                        if (flagIn) {
                            flag[each] = flagIn;
                            break;
                        }
                    }
                }
                bool getAns = true;
                for (int i = 0; i < (int)flag.size(); i++)
                    if (!flag[i]) {
                        getAns = false;
                        break;
                    }
                if (getAns)
```

```
                return true;// 全部存在一致的表达式
            }
        }

        return false;

    }

    // @function : 长度限制归结
    bool clauseResolution::lengthDeal(CLAUSES& c, vector<CLAUSES>& tempBag) {
        if ((int)c.expressions.size() > LENGTHEDGE)
            return true;
        return false;
    }

    // @function : 子句归结
    void clauseResolution::resolution(void) {

        //initialShow(); // 展示输入读取数据 for test
        initialRecord(); // 初始输入数据存储
        int lastSite = 0;

        while (true) {

            // 遍历初始化
            int len = clauseBag.size();
            vector<CLAUSES>clauseTempBag;

            // 在现有的clauseBag内两两配对计算
            for (int i = 0; i < len; i++) {
                for (int j = i + 1; j < len; j++) {
                    if (j >= lastSite) {
                        CLAUSES ansClause;       // 归结结果
                        vector<char> record;     // 过程记录
                        if (twoClauseResolu(clauseBag[i], clauseBag[j], ansClause, record)) {

                            vector<char>upData;  // 更迭过程存储
                            RECORD temp;         // 结果存储

                            // 新子句更新
                            clausesUpdate(ansClause, record, upData);

                            // 存储更迭结果
                            processRecord(clauseBag[i], clauseBag[j], ansClause, upData, temp);

                            // 判断是否得到结果
                            if (judgeAnswer(ansClause)) {

                                // 得到最终结果
                                answer = ansClause;

                                // 显示本次更迭
                                //processShow(clauseBag[i], clauseBag[j], ansClause, upData);
    // for test
```

```
                                // 数据结果存储
                                clauseTempBag.push_back(ansClause);
                                for (int i = 0; i < (int)clauseTempBag.size(); i++)
                                    clauseBag.push_back(clauseTempBag[i]);

                                // 流程记录的添加
                                recordList.push_back(temp);

                                // 显示最终结果
                                //cout << "answer:" << answer.expressions[0].element[0] << endl;
// for test
                                //tempShow();

                                return;
                            }
                            // 包含归结
                            else if (containDeal(ansClause, clauseTempBag))
                                --countNum;
                            // 长度限制
                            else if (lengthDeal(ansClause, clauseTempBag))
                                --countNum;
                            // 等式归结
                            else if (equalDeal(ansClause, clauseTempBag)) {
                                // 过程记录添加
                                recordList.push_back(temp);
                            }
                            // 子句更迭
                            else {
                                // 流程记录的添加
                                recordList.push_back(temp);

                                // 新子句添加
                                clauseTempBag.push_back(ansClause);

                                // 显示本次更迭
                                //processShow(clauseBag[i], clauseBag[j], ansClause,
upData);// for test
                            }
                        }
                    }
                }
            }
            lastSite = len;// 记录上次长度 避免重复计算

            // 加入归结产生的新子句
            for (int i = 0; i < (int)clauseTempBag.size(); i++)
                clauseBag.push_back(clauseTempBag[i]);
        }
    }

    // @function : 回溯最优解的过程
    void clauseResolution::traceBack(void) {
        string lastName = answer.name;
```

```
    for (int i = 0; i < (int)recordList.size(); i++)
        if (recordList[i].resultName == answer.name)
            traceBackIn(recordList[i]);
    return;
}

// @function : 寻找最优解的回溯函数
void clauseResolution::traceBackIn(RECORD& c) {

    // father1_get
    for (int i = 0; i < (int)recordList.size(); i++) {
        if (recordList[i].resultName == c.father1Name)
            traceBackIn(recordList[i]);
    }
    // father2_get
    for (int i = 0; i < (int)recordList.size(); i++) {
        if (recordList[i].resultName == c.father2Name)
            traceBackIn(recordList[i]);
    }
    // ans_get
    for (int i = 0; i < (int)recordList.size(); i++) {
        if (recordList[i].resultName == c.resultName)
            bestPath.push_back(recordList[i]);
    }
    return;
}

// @function : 记录起始过程
void clauseResolution::initialRecord(void) {
    for (int outerI = 0; outerI < (int)clauseBag.size(); outerI++) {
        RECORD temp;
        temp.change = "";
        temp.father1Name = "";
        temp.father2Name = "";

        temp.resultName = clauseBag[outerI].name;
        temp.result = "";

        for (int i = 0; i < (int)clauseBag[outerI].expressions.size(); i++) {
            if (clauseBag[outerI].expressions[i].IsNot == false)
                temp.result += "~";
            temp.result += clauseBag[outerI].expressions[i].type;
            temp.result += "(";
            for (int j = 0; j < (int)clauseBag[outerI].expressions[i].element.size(); j++) {
                if (clauseBag[outerI].expressions[i].element[j] != FUNCTION)
                    temp.result += clauseBag[outerI].expressions[i].element[j];
                else
                    temp.result = temp.result + clauseBag[outerI].expressions[i].func.name +
"(" + clauseBag[outerI].expressions[i].func.parameter + ")";
                if (j != (int)clauseBag[outerI].expressions[i].element.size() - 1)
                    temp.result += ",";
            }
            temp.result += ")";
```

```
            if (i != (int)clauseBag[outerI].expressions.size() - 1)
                temp.result += "|";
        }

        initialList.push_back(temp);
    }
    return;
}

// @function : 记录存储过程
void clauseResolution::processRecord(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>&
upData, RECORD& get) {

    // 存储结果表达式
    get.resultName = ans.name;
    get.result = "";
    for (int i = 0; i < (int)ans.expressions.size(); i++) {
        if (ans.expressions[i].IsNot == false)
            get.result+= "~";
        get.result += ans.expressions[i].type;
        get.result += "(";
        for (int j = 0; j < (int)ans.expressions[i].element.size(); j++) {
            if (ans.expressions[i].element[j] != FUNCTION)
                get.result += ans.expressions[i].element[j];
            else
                get.result =  get.result + ans.expressions[i].func.name + "(" +
ans.expressions[i].func.parameter + ")";
            if (j != (int)ans.expressions[i].element.size() - 1)
                get.result += ",";
        }
        get.result += ")";

        if (i != (int)ans.expressions.size() - 1)
            get.result += "|";
    }

    // 存储递归来源
    get.father1Name = c1.name;
    get.father2Name = c2.name;

    // 存储递归替换
    get.change = "";
    for (int i = 0; i < (int)upData.size();) {
        if (upData[i] != FUNCTION) {
            get.change+= upData[i];
            i += 1;
        }
        else {
            get.change = get.change + upData[i + 1] + "(" + upData[i + 2] + ")";
            i += 3;
        }
        get.change += "-->";
        if (upData[i] != FUNCTION) {
```

```
                get.change += upData[i];
                i += 1;
            }
            else {
                get.change = get.change + upData[i + 1] + "(" + upData[i + 2] + ")";
                i += 3;
            }
            if (i < (int)upData.size() - 1)
                get.change += ";";
        }

        return;
}

// @function : 显示归结过程
void clauseResolution::allShow(void) {
        // 原始逻辑显示
        for (int i = 0; i < (int)initialList.size(); i++) {
            cout << left << setw(8)<< initialList[i].resultName;
            cout << left << setw(30) << initialList[i].result << endl;
        }
        // 归结过程显示
        for (int i = 0; i < (int)recordList.size(); i++) {
            cout << left << setw(8) << recordList[i].resultName;
            cout << left << setw(30) << recordList[i].result;
            cout << left << setw(15) << recordList[i].father1Name + '+' + recordList[i].father2Name;
            cout << left << setw(15) << recordList[i].change << endl;
        }
        return;
}

// @function : 显示最优归结
void clauseResolution::bestShow(void) {
        // 原始逻辑显示
        for (int i = 0; i < (int)initialList.size(); i++) {
            cout << left << setw(8) << initialList[i].resultName;
            cout << left << setw(30) << initialList[i].result << endl;
        }
        // 归结过程显示
        for (int i = 0; i < (int)bestPath.size(); i++) {
            cout << left << setw(8) << bestPath[i].resultName;
            cout << left << setw(30) << bestPath[i].result;
            cout << left << setw(15) << bestPath[i].father1Name + '+' + bestPath[i].father2Name;
            cout << left << setw(15) << bestPath[i].change << endl;
        }
        return;
}

// @function : 显示推断结果
void clauseResolution::ansShow(void) {
        cout << "谋杀A的人是:" << answer.expressions[0].element[0] << endl;
        return;
}
```

```
// ----------------------------
// | @NOTICE   : 以下用于测试  |
// ----------------------------
// @function : 打印归结过程
void clauseResolution::processShow(CLAUSES& c1, CLAUSES& c2, CLAUSES& ans, vector<char>& upData)
{

    // 表达式显示
    cout << ans.name << " : ";
    for (int i = 0;i < (int)ans.expressions.size(); i++) {
        if (ans.expressions[i].IsNot == false)
            cout << "~";
        cout << ans.expressions[i].type;
        cout << "(";
        for (int j = 0; j < (int)ans.expressions[i].element.size(); j++) {
            if (ans.expressions[i].element[j] != FUNCTION)
                cout << ans.expressions[i].element[j];
            else
                cout << ans.expressions[i].func.name << "(" <<
ans.expressions[i].func.parameter << ")";
            if (j != (int)ans.expressions[i].element.size() - 1)
                cout << ",";
        }
        cout << ")";

        if (i != (int)ans.expressions.size() - 1)
            cout << "|";
    }

    // 显示递归来源
    cout << "   ";
    cout << "{{" << c1.name << "与" << c2.name << "归结";
    if (upData.size() != 0)
        cout << " 且 ";
    for (int i = 0; i < (int)upData.size();) {
        if (upData[i] != FUNCTION) {
            cout << upData[i];
            i += 1;
        }
        else {
            cout << upData[i + 1] << "(" << upData[i + 2] << ")";
            i += 3;
        }
        cout << "-->";
        if (upData[i] != FUNCTION) {
            cout << upData[i];
            i += 1;
        }
        else {
            cout << upData[i + 1] << "(" << upData[i + 2] << ")";
            i += 3;
        }
        if (i < (int)upData.size() - 1)
```

```
            cout << ";";
        }
        cout << "}}" << endl;

        return;
}
// @function : 打印初始逻辑
void clauseResolution::initialShow(void) {

        for (int outerI = 0; outerI < (int)clauseBag.size(); outerI++) {
            cout << clauseBag[outerI].name << " : ";
            for (int i = 0; i < (int)clauseBag[outerI].expressions.size(); i++) {
                if (clauseBag[outerI].expressions[i].IsNot == false)
                    cout << "~";
                cout << clauseBag[outerI].expressions[i].type;
                cout << "(";
                for (int j = 0; j < (int)clauseBag[outerI].expressions[i].element.size(); j++) {
                    if (clauseBag[outerI].expressions[i].element[j] != FUNCTION)
                        cout << clauseBag[outerI].expressions[i].element[j];
                    else
                        cout << clauseBag[outerI].expressions[i].func.name << "(" <<
clauseBag[outerI].expressions[i].func.parameter << ")";
                    if (j != (int)clauseBag[outerI].expressions[i].element.size() - 1)
                        cout << ",";
                }
                cout << ")";

                if (i != (int)clauseBag[outerI].expressions.size() - 1)
                    cout << "|";
            }

            cout << endl;
        }
        return;
}
// @function : 临时展示clauseBag数据 --> for test
void clauseResolution::tempShow(void) {
        //for (int i = 0; i < (int)clauseBag.size(); i++) {

        //  // 输出子句名
        //  cout << clauseBag[i].name<<"--";

        //  // 输出子句中间步骤
        //  for (int j = 0; j < (int)clauseBag[i].expressions.size(); j++) {
        //      cout << clauseBag[i].expressions[j].IsNot << '-';
        //      cout << clauseBag[i].expressions[j].type << '-';
        //      for (int k = 0; k < (int)clauseBag[i].expressions[j].element.size(); k++) {
        //          if (clauseBag[i].expressions[j].element[k] != FUNCTION)
        //              cout << clauseBag[i].expressions[j].element[k] << '-';
        //          else
        //              cout << clauseBag[i].expressions[j].func.name << "(" <<
clauseBag[i].expressions[j].func.parameter << ")";
        //      }
```

```
    //        cout << "***";
    //    }
    //    cout << endl;
    //}

    //// all record for test
    //for (int i = 0; i < (int)initialList.size(); i++) {
    //    cout << initialList[i].resultName << " " << initialList[i].result << endl;
    //}
    //for (int i = 0; i < (int)recordList.size(); i++) {
    //    cout << recordList[i].resultName << ":" << recordList[i].result << "***";
    //    cout << recordList[i].father1Name << "-" << recordList[i].father2Name << "***" <<
recordList[i].change << endl;
    //}

    // best record show
    //for (int i = 0; i < (int)bestPath.size(); i++) {
    //    cout << bestPath[i].resultName << ":" << bestPath[i].result << "***";
    //    cout << bestPath[i].father1Name << "-" << bestPath[i].father2Name << "***" <<
bestPath[i].change << endl;
    //}

    return;
}



/*****************************************************************************************/
/*************************************工具函数*********************************************/
/*****************************************************************************************/

// @function : 读取数据
void dataIn(vector<string>& originalSentence) {
    string dataIn;
    while (true) {
        cin >> dataIn;
        if (dataIn == "#")
            break;
        else
            originalSentence.push_back(dataIn);
    }
    return;

}
```

logicalReasoning_ui.cpp

```
/*
 * @author   : gonzalez
 * @time     : 2022.5.26-2022.6.2
 * @function :
 */

#include"logicalReasoning_head.h"
```

```
using namespace std;

// @function : clauseUI类初始化
clauseUI::clauseUI(void) {

    // 照片初始化
    loadimage(&background, _T("./image/background.png"), UILENGTH, UIWIDTH, false);

    loadimage(&input, _T("./image/input1.png"), BUTTONLENGTH, BUTTONWIDTH, false);
    loadimage(&inputActive, _T("./image/input2.png"), BUTTONLENGTH, BUTTONWIDTH, false);
    loadimage(&showAll, _T("./image/showAll1.png"), BUTTONLENGTH, BUTTONWIDTH, false);
    loadimage(&showAllActive, _T("./image/showAll2.png"), BUTTONLENGTH, BUTTONWIDTH, false);
    loadimage(&showBest, _T("./image/showBest1.png"), BUTTONLENGTH, BUTTONWIDTH, false);
    loadimage(&showBestActive, _T("./image/showBest2.png"), BUTTONLENGTH, BUTTONWIDTH, false);

    loadimage(&restart, _T("./image/restart1.png"), RESTARTLEN, RESTARTLEN, false);
    loadimage(&restartActive, _T("./image/restart2.png"), RESTARTLEN, RESTARTLEN, false);

    // 全景区域
    backArea.left = 0;
    backArea.top = 0;

    // 输入按钮区域
    inputArea.left = 500;
    inputArea.top = 70;

    // 展示归结过程按钮区域
    showallArea.left = 500;
    showallArea.top = 165;

    // 展示最优归结路径按钮区域
    showbestArea.left = 500;
    showbestArea.top = 260;

    // 重启按钮区域
    restartArea.left = 702;
    restartArea.top = 344;

}


// @function : 展示背景
void clauseUI::bgShow(void) {
    putimage(backArea.left, backArea.top, &background);
    putimage(inputArea.left, inputArea.top, &input);
    putimage(showallArea.left, showallArea.top, &showAll);
    putimage(showbestArea.left, showbestArea.top, &showBest);
    putimage(restartArea.left, restartArea.top, &restart);
    return;
}


// @function : 鼠标操作事件
```

```
int clauseUI::orderChoose(bool haveIn) {
    ExMessage msg;
    while (true) {
        //鼠标事件
        if (peekmessage(&msg, EM_MOUSE)) {
            // 鼠标点击
            if (msg.message == WM_LBUTTONDOWN) {
                if ((msg.x >= inputArea.left && msg.x <= inputArea.left + BUTTONLENGTH)
                    && (msg.y >= inputArea.top && msg.y <= inputArea.top + BUTTONWIDTH)) {
                    if (!haveIn) {
                        putimage(inputArea.left, inputArea.top, &input);
                        return MOUSEINPUT;// 输入
                    }
                }
                if ((msg.x >= showallArea.left && msg.x <= showallArea.left + BUTTONLENGTH)
                    && (msg.y >= showallArea.top && msg.y <= showallArea.top + BUTTONWIDTH))
{
                    putimage(showallArea.left, showallArea.top, &showAll);
                    return MOUSEALL;// 展示所有
                }
                if ((msg.x >= showbestArea.left && msg.x <= showbestArea.left + BUTTONLENGTH)
                    && (msg.y >= showbestArea.top && msg.y <= showbestArea.top + BUTTONWIDTH))
{
                    putimage(showbestArea.left, showbestArea.top, &showBest);
                    return MOUSEBEST;// 展示最佳
                }
                if ((msg.x >= restartArea.left && msg.x <= restartArea.left + RESTARTLEN)
                    && (msg.y >= restartArea.top && msg.y <= restartArea.top + RESTARTLEN))
{
                    putimage(restartArea.left, restartArea.top, &restart);
                    return MOUSEREST;// 重启
                }
            }
            // 鼠标移动
            else if (msg.message == WM_MOUSEMOVE) {
                if ((msg.x >= inputArea.left && msg.x <= inputArea.left + BUTTONLENGTH)
                    && (msg.y >= inputArea.top && msg.y <= inputArea.top + BUTTONWIDTH)) {
                    if (!haveIn)
                        putimage(inputArea.left, inputArea.top, &inputActive);
                }
                else if ((msg.x >= showallArea.left && msg.x <= showallArea.left +
BUTTONLENGTH)
                    && (msg.y >= showallArea.top && msg.y <= showallArea.top + BUTTONWIDTH))
{
                    putimage(showallArea.left, showallArea.top, &showAllActive);
                }
                else if ((msg.x >= showbestArea.left && msg.x <= showbestArea.left +
BUTTONLENGTH)
                    && (msg.y >= showbestArea.top && msg.y <= showbestArea.top + BUTTONWIDTH))
{
                    putimage(showbestArea.left, showbestArea.top, &showBestActive);
                }
                else if ((msg.x >= restartArea.left && msg.x <= restartArea.left + RESTARTLEN)
                    && (msg.y >= restartArea.top && msg.y <= restartArea.top + RESTARTLEN))
```

```
{
                    putimage(restartArea.left, restartArea.top, &restartActive);
                }
                else {
                    putimage(inputArea.left, inputArea.top, &input);
                    putimage(showallArea.left, showallArea.top, &showAll);
                    putimage(showbestArea.left, showbestArea.top, &showBest);
                    putimage(restartArea.left, restartArea.top, &restart);
                }
            }
        }
    }
}
```

logicalReasoning_main.cpp

```
/*
 * @author   : gonzalez
 * @time     : 2022.5.26-2022.6.2
 * @function :
 */

#include "logicalReasoning_head.h"
using namespace std;


int main()
{
    initgraph(UILENGTH, UIWIDTH, EW_SHOWCONSOLE);          //初始化图形界面
    setbkcolor(BLACK);                                     //设置背景颜色
    clauseUI ui;
    ui.bgShow();
    while (true) {
        // initiate
        clauseResolution c;
        vector<string> originalSentence;
        bool haveIn = false;
        // input
        while (true) {
            if (ui.orderChoose() == MOUSEINPUT) {
                haveIn = true;
                cout << "请输入推断子句集（notice:请以#结尾）" << endl;
                dataIn(originalSentence);
                cout << "输入结束" << endl;
                Sleep(1000);
                break;
            }
        }
        // operate
        c.clausInput(originalSentence);
        c.resolution();
        c.traceBack();
```

```
        // show
        while (true) {
            int getOrder = ui.orderChoose(haveIn);
            if (getOrder == MOUSEALL) {
                system("cls");
                cout << "归结过程如下:" << endl;
                Sleep(1000);
                c.allShow();
                c.ansShow();
            }
            else if (getOrder == MOUSEBEST) {
                system("cls");
                cout << "最优过程如下:" << endl;
                Sleep(1000);
                c.bestShow();
                c.ansShow();
            }
            else if (getOrder == MOUSEREST) {
                system("cls");
                break;
            }
        }
        system("cls");
    }


    return 0;
}
```