

**课程设计报告**

**题目：基于SAT的数独游戏求解程序**

**课程名称：程序设计综合课程设计**

**专业班级： 计算机2009班**

**学 号： U202012656**

**姓 名： 张锦程**

**指导教师： 向文**

**报告日期： 2021年9月13日**

**计算机科学与技术学院**

**目录**

**任务书** I

**1引言** 1

1.1 课题背景与意义 1

* + 1. SAT 1
    2. 数独游戏 1

1.2 国内外研究现状 2

1.3课程设计的主要研究工作 3

**2系统需求分析与总体设计** 4

2.1 系统需求分析 4

2.2 系统总体设计 4

**3系统详细设计** 6

3.1有关数据结构的定义 6

3.2主要算法设计 8

3.2.1 CNF文件处理 8

3.2.2 DPLL算法处理 9

3.2.3 Sudoku算法处理 16

**4系统实现与测试** 18

4.1系统实现 18

4.1.1 软硬件环境 18

4.1.2 数据类型定义 18

4.1.2 函数声明以及函数调用关系 20

4.2系统测试 23

4.2.1 交互系统展示 23

4.2.2 CNF文件处理模块测试和DPLL算法模块测试 23

4.2.3 CNF算例测试总结表格 47

4.2.4 数独游戏求解及简易游玩模块测试 50

**5总结与展望** 54

5.1全文总结 54

5.2工作展望 54

**6体会** 55

**参考文献** 56

**附录** 57

**任务书**

**设计内容**

SAT问题即命题逻辑公式的可满足性问题（satisfiability problem），是计算机科学与人工智能基本问题，是一个典型的NP完全问题，可广泛应用于许多实际问题如硬件设计、安全协议验证等，具有重要理论意义与应用价值。本设计要求基于DPLL算法实现一个完备SAT求解器，对输入的CNF范式算例文件，解析并建立其内部表示；精心设计问题中变元、文字、子句、公式等有效的物理存储结构以及一定的分支变元处理策略，使求解器具有优化的执行性能；对一定规模的算例能有效求解，输出与文件保存求解结果，统计求解时间。

**设计要求**

要求具有如下功能：

1. **输入输出功能：**包括程序执行参数的输入，SAT算例cnf文件的读取，执行结果的输出与文件保存等。(15%)
2. **公式解析与验证：**读取cnf算例文件，解析文件，基于一定的物理结构，建立公式的内部表示；并实现对解析正确性的验证功能，即遍历内部结构逐行输出与显示每个子句，与输入算例对比可人工判断解析功能的正确性。数据结构的设计可参考文献[1-3]。(15%)
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。(35%)
4. **时间性能的测量：**基于相应的时间处理函数（参考time.h），记录DPLL过程执行时间（以毫秒为单位），并作为输出信息的一部分。(5%)
5. **程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略[1-3]等某一方面进行优化设计与实现，提供较明确的性能优化率结果。优化率的计算公式为：[(t-to)/t]\*100%,其中t 为未对DPLL优化时求解基准算例的执行时间，to则为优化DPLL实现时求解同一算例的执行时间。(15%)
6. **SAT应用：**将数独游戏[5]问题转化为SAT问题[6-8]，并集成到上面的求解器进行数独游戏求解，游戏可玩，具有一定的/简单的交互性。应用问题归约为SAT问题的具体方法可参考文献[3]与[6-8]。(15%)

**1引言**

**1.1课题背景与意义**

#### 1.1.1 SAT

SAT问题即布尔可满足性问题，是确定是否存在满足给定布尔公式解释的问题。如果对于给定的布尔公式变量可以一致地用TRUE或者FALSE替换，那么该布尔公式可满足，相反则不满足。

SAT问题是计算机科学与人工智能领域的经典问题，研究成果广泛应用于电子设计自动化，人工智能等领域，因此研究SAT问题有助于拓展知识面以及今后的实际应用。

**1.1.2数独游戏**

数独游戏的历史渊源比较久远，数独是一种源自18世纪末的瑞士，后在美国发展并在日本得以发扬光大的数学智力拼图游戏。

早在数千年前，中国人就发明了[九宫图](https://baike.so.com/doc/4990670-5214413.html" \t "https://baike.so.com/doc/_blank):在9个方格中，横行和竖行的数字总和是相同的。"数独"也不是什么新生事物，已经存在了数百年。18世纪，瑞士数学家莱昂哈德·欧勒发明了"拉丁方块"，但并没有受到人们的重视。直到20世纪70年代，美国杂志才以"数字拼图"的名称将它重新推出。日本随后接受并推广了这种游戏，并且将它改名为"数独"，大致的意思是"独个的数字"或"只出现一次的数字"。

现今流行的数独于1984年由日本游戏杂志《パズル通信ニコリ》发表并得了现时的名称。数独本是"独立的数字"的省略，因为每一个方格都填上一个非零的个位数。数独冲出日本成为英国当下的流行游戏，得归功于曾任[香港高等法院](https://baike.so.com/doc/5677194-5889867.html" \t "https://baike.so.com/doc/_blank)法官的高乐德(Wayne Gould)。2004年，他在日本旅行的时候，发现杂志上介绍的这款游戏，便带回伦敦向《泰晤士报》推介并获得接纳。英国《每日邮报》也于三日后开始连载，使数独在英国正式掀起热潮。数独不仅是报章增加销量的法宝，脑筋动得快的《泰晤士报》还做起手机族的生意，花4.5英镑就能下载10则数独游戏到手机上玩。渐渐，其他国家和地区受其影响也开始风靡数独。

同类似的填字游戏不同，数独受欢迎的原因之一是它既不需要丰富的百科知

识，也不要掌握大量的词汇，这使其能迅速为孩子和初学者所接受。根据游戏开始时的方格中已有的数字和位置，数独难易程度不同，有些复杂的甚至令数学家也不能完成。据著名的动游戏开发商Astraware Ltd.预计，移动数独游戏的版本多达几十种，Palm和Windows Mobile设备版本的数独游戏就各有20种左右。Sudokumo推出的移动数独游戏，能够下载到大多数手机中。这家位于英国的游戏软件公司表示，已经在全球卖出了7500套数独游戏，而且来自用户的兴趣还在增加。

**1.2国内外研究现状**

求解SAT问题的经典算法——Dpll算法，它在1962年由马丁·戴维斯、希拉里·普特南、乔治·洛吉曼和多纳·洛夫兰德共同提出，作为早期戴维斯-普特南算法的一种改进。戴维斯-普特南算法是戴维斯与普特南在1960年发展的一种算法。DPLL是一种高效的程序，并且经过40多年还是最有效的SAT解法，以及很多一阶逻辑的自动定理证明的基础。

在之后Bart Selman和Henry Kautz分别于1997年和2003年在人工智能第五届国际合作会议上提出了SAT问题面临的十大挑战性问题，并在2001年和2007年先后 对当时的可满足性问题现状进行了全面的阐述和总结。这十大挑战性问题的提出对SAT基准问题的理论研究和算法改进都起到了强有力的推动作用。这使得越来越多的人开始关注并研究SAT问题，所以这段时间也涌现出了众多新的高效的SAT算法如MINISAT、SATO、CHAFF、POSIT和GRASP等，SAT算法的研究成果显著，求解算法也越来越多地应用到了实际问题领域。这些 新兴的算法大都是基于DPLL算法的改进算法，改进的方面包括：采用新的数据结构、新的变量决策策略或者新的快速的算法实现方案。国内也涌现出了许多高效的求解算法，如1998年梁东敏提出了改进的子句加权WSAT算法，2000年金人超和黄文奇提出了并行Solar算法，2002年张德富提出了模拟退火算法。

SAT国际竞赛从2002年开始每隔一到两年举办一次，这也极大地推动了SAT问题的研究，由此可见SAT求解问题仍在继续被人们所探索。

**1.3课程设计的主要研究工作**

1. 首先对DPLL算法，SAT求解问题的背景，原理进行深入了解，根据相关资料对于项目做一个整体的设计
2. 设计相应数据结构与算法来完成基于DPLL的SAT求解器的实现，并用提供的算例做相应测试；

设计程序要求模块化，程序源代码进行模块化组织。主要模块包括如下：

1. 主控、交互与显示模块（display）
2. CNF解析模块（cnfparser）
3. 核心DPLL模块( solver)
4. 数独模块,包括游戏格局生成、归约、求解(sudoku)
5. 从改变存储结构或者选取文字策略等方面来实现算法的优化，设计测试方案来总结优化的效果；
6. 设计问题转化策略将数独问题归约为SAT问题并求解；
7. 将各个分支整合为项目并调试直到能正确运行。

**2系统需求分析与总体设计**

**2.1系统需求分析**

基于DPLL算法的SAT求解器可以求解部分布尔算例，可以对CNF文件进行解析和结果输出；  
 可以创建随机的数独游戏并且具有一定的交互性，可以用来求解和尝试可视化自主求解数独游戏。

**2.2系统总体设计**

系统总体设计分为两个大的模块：基于DPLL算法的SAT求解器和简易数独游戏，各自模块下面还有一些小的功能，大致介绍如下：

1. 基于DPLL算法的SAT求解器，能够完成如下功能：

（1） CNF的读取解析，遍历输出，保存；

（2） DPLL求解，计算求解时间并显示，将结果保存到同名.res文件里。

2. 数独游戏，能够完成如下功能：

（1） 随机数独游戏的创建（有难度选择）,并可以实现一定程度上的交互；

（2） 读入数独游戏文件（txt），转化为CNF文件DPLL进行求解，再可视化地将结果打印到屏幕上；

系统总体设计流程图如下图2.1所示：

图2.1参考程序流程图

**3系统详细设计**

**3.1有关数据结构的定义**

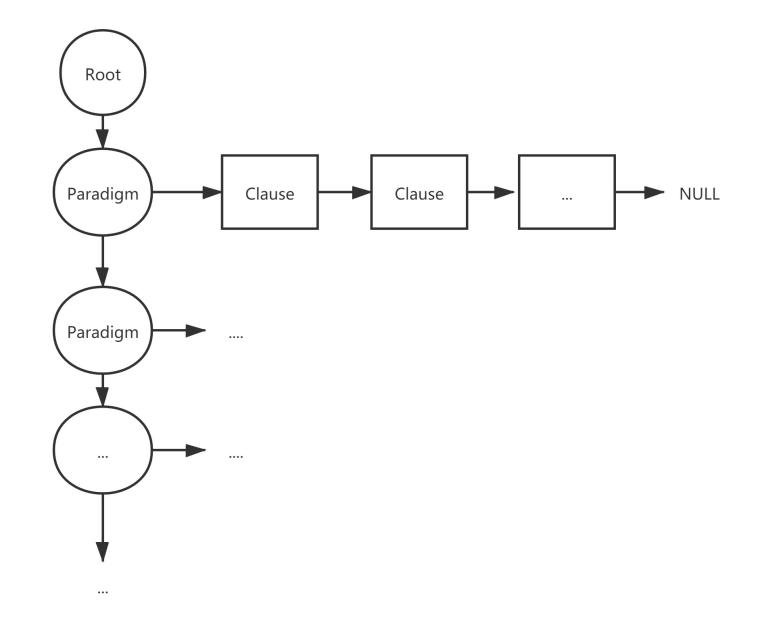
1. 首先是对CNF文件进行读取，就要处理CNF文件中的文字和子句：
2. 对于文字需要用一个结构体Clause来定义：数据项有int型的文字值，int型用于标记删除的flag，指向下一个文字的指针nextl以及后续优化时遍历查找的整个CNF公式邻接链表中下一个文字相同的子句结点litline，用于回复与方便查找；
3. 对于子句需要用一个结构体Paradigm来定义：数据项有int型的number表示该子句的文字数；int型的flag表示标记删除；指向下一个子句的指针nextc以及指向文字的头指针sentence；
4. 用一个结构体Root来定义CNF范式链表头结点类型，用于存储CNF范式信息：数据项有int型的litsize用于存储文字数量；int型的parasize用于存储子句数量以及指向第一个子句的指针first；

表3-1对CNF文件读取和处理相关结构体

|  |  |  |
| --- | --- | --- |
| 数据名 | 用途 | 数据项 |
| Clause | 文字 | int：literal，flag；  struct Clause \*：nextl，litline； |
| Paradigm | 子句 | int： number，flag;  struct Paradigm \*nextc;  struct Clause \*sentence; |
| Root | 头节点 | int litsize，parasize  Paradigm \*first |

关于数据结构的图直观表示，如图3.2表示：

（还有指向相同文字的指针不易直观表示）



1. 其次是DPLL算法中，需要对文字进行查找，就要遍历以及记忆：
2. 定义指向子句链表头结点的指针链表结点结构类型Paraline：数据项有指向子句链表头结点Paradigm的claline；指向下一链表结点的next；（优化）
3. 定义文字相关信息链表结构类型LitTrabverse：数据项有指向含有该正文字或负文字的子句头结点链表的头结点的Tra\_cla，指向该正文字或负文字的文字结点的Tra\_lit；（优化）
4. 存储变元信息的变元线性表结点结构ArgueValue：int型的Value表示储存变元的真值；int型的IsInit表示变元是否已赋值；int型的Occur\_times表示变元在所有子句中出现的总次数，用于分裂政策的选择（优化）；LitTrabverse型的变元所有正文字的相关信息结构Pos（优化）和变元所有负文字的相关信息结构Neg（优化）；sudokusolver型的求解数独文件时所需的变元信息的xyz；

表3-2 DPLL算法处理相关结构体

|  |  |  |
| --- | --- | --- |
| 数据名 | 用途 | 数据项 |
| Paraline | 定义指向子句链表头结点的指针链表结点结构类型 | Paradigm \*claline  struct Paraline \*next; |
| LitTrabverse | 定义文字相关信息链表结构类型 | Paraline \*Tra\_cla;  Clause \*Tra\_lit; |
| ArgueValue | 定义存储变元信息的变元线性表结点结构类型 | int Value;//变元的真值  int IsInit;//变元是否赋值  int Occur\_times;//变元在所有子句中出现的总次数，用于分裂政策的选择  LitTrabverse Pos;//变元所有正文字的相关信息结构  LitTrabverse Neg;//变元所有负文字的相关信息结构  sudokusolver xyz;//求解数独文件时所需的变元信息 |

1. 最后是数独游戏增加的结构sudokusolver用于储存行列以及值： int 的x存储行信息；int的y存储列信息；int的z存储变元对应1～9数值信息；

**3.2主要算法设计**

#### 3.2.1 CNF文件处理

#### 创建子句链表及文字链表：

#### int CreateClause(FILE \*\*fp,Clause \*\*sentence,Paradigm \*ClausHead,int first);

参数：(FILE\*\*)open：文件的指针地址

(Clause \*\*)sentence:子句的头结点 Clause类型指针地址

(Paradigm \*)ClausHead：子句链表头节点指针值

(int )first：创建的子句的第一个文字的值

返回值：i：子句内文字的数量

ERROR：该子句为空子句

OVERFLOW：空间不足溢出

读入一个值后，创建子句节点进行赋值，随后移动继续添加；关键在于不要遗漏正文字和负文字的子句链表添加以及变元真值表的变元出现次数增加，后续的DPLL操作会用到，也是优化程序的关键步骤；

1. 创建CNF范式邻接链表及变元表

status CreateParadigm(FILE \*\*fp);

参数：(FILE\*\*)open：文件的指针地址

读入CNF文件，首先要区分c和p，c不读入，p开始读入，同时记录文字数和子句数，根据文字数初始化变元真值表，随后循环创建子句以及构建正文字和负文字的子句链表，优化时创建数组来降序排列变元的出现次数，进而分裂原则选择文字进行优化；

1. 销毁所有链表及线性表结构

status DestroyParadigm(Root \*r);

用while（）循环销毁创建的子句链表Root，再释放变元正负文字信息链表存储空间。

1. 判断CNF范式中是否还含有单子句

int HasUnitClause(Root \*r);

返回类型：int

返回值：存在单子句：该单子句所含的唯一文字

不存在单子句：0

进入子句链表后只要判断子句头节点的number是否为1即可判断是否为单子句，再返回文字值；

1. 判断CNF范式中是否还含有单子句（优化前版本）

Clause \* HasUnitClause\_Before(Root \*r);

返回类型：Clause \*

返回值：存在单子句：该单子句所含的唯一Clause结点的指针

不存在单子句：NULL

进入子句链表后只要判断子句头节点的number是否为1即可判断是否为单子句，再返回指针值；

1. 判断参数指针c指向的子句头结点对应子句是否为单子句；

status isUnitClause(Paradigm \*c);

返回值：TRUE：是单子句；

ERROR：不是单子句

只要判断该子句头节点的number是否为1即可判断是否为单子句；

#### 3.2.2 Dpll算法处理

/\*在整个CNF公式中取一个文字\*/

1. 取每次DPLL处理后公式中Occur\_Times最大的文字

int FindLiteral1(Root \*r);

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0

找到变元表正序第一个未被赋真值的变元，num记录其序号，并用flag记录已经取了未被赋真值置为1，否则为0；然后遍历真值表，找到Occur\_times最大的值的输出。

1. 取原公式中未赋真值的变元中出现次数最多（Occur\_Times最大）的变元正文字

int FindLiteral2(Root \*r);

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0；

根据之前降序排序的Frequency数组储存原公式的变元次数，遍历查找已经未赋值的IsInit，再返回；

1. 取子句中第一个未被删除（flag为0）的文字

int FindLiteral3(Root \*r);

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0

进入子句链表，for()遍历进入子句头节点，再for()遍历进入子句内部节点，找到子句链表节点flag为0的链表节点返回文字值；

1. 取变元表正序正序第一个未赋真值（IsInit=0）的变元的正文字

int FindLiteral3(Root \*r);

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0

遍历真值表中的正文字链表即可。

1. 在整个CNF公式中取一个文字（优化前版本）：取子句中第一个未被删除（flag为0）的文字

Clause \* FindLiteral\_Before(Root \*r);

返回类型：Clause \*

返回值：公式非空：选中文字的结点指针

公式中没有剩余文字：NULL

进入子句链表，for()遍历进入子句头节点，再for()遍历进入子句内部节点，找到子句链表节点flag为0的链表节点返回指针；

/\*单子句原则\*/

1. 删除出现了文字l的所有单子句

status DeleteClause(Root \*r,int l);

参数l：真值为1的文字（int）

(此函数不应使公式中的子句数量减少）

先判断l的正负值，如果l为正文字情况，搜索l对应变元的正文字信息链表，进入子句链表同时p->flag=abs（l）表明因为l而删除；将该子句的所有的链表节点flag=abs（l）表明因为l而删除。与此同时子句数，文字数和变元次数都相应减少。负文字情况同理。

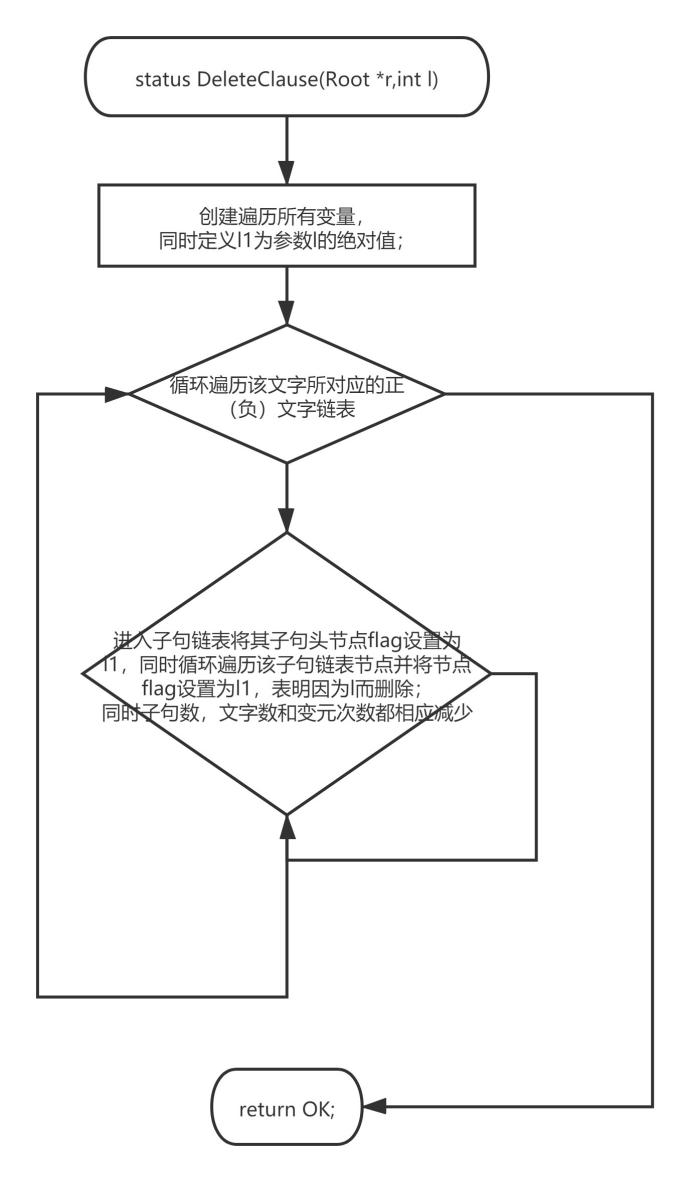


图3.2-1 删除出现了文字l的所有单子句

1. 删除所有文字为-l的子句链表结点

status DeleteLiteral(Root \*r,int l);

参数l：(int)真值为1的文字；

返回值：OK：成功删除文字；

FALSE：公式无解

(此函数不应使公式中的子句数量减少）

先判断l的正负值，如果l为正文字情况，搜索l对应变元的正文字信息链表，进入子句链表；将该子句的找到对应-l文字值的链表节点，表明因为l而删除。如果子句文字数为1，则删除失败返回ERROR，否则链表节点的flag的值置为abs（l）；与此同时该子句文字数和变元次数都相应减少。负文字情况同理可得。

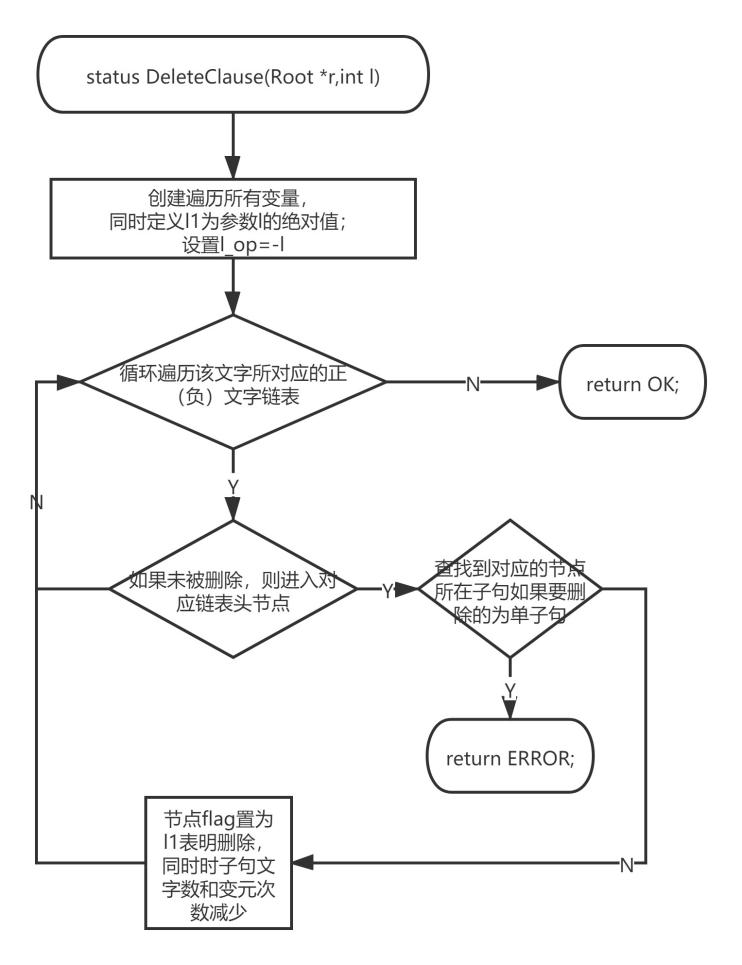


图3.2-2 删除所有文字为-l的子句链表结点

1. CNF范式邻接链表表头添加只含有文字l的单子句链表

status AddClause(Root \*r,int l);

参数l：(int)增加的单子句内的唯一文字

返回值：OVERFLOW：空间不足溢出；

OK：子句增加成功

用malloc函数分配内存空间之后移动r->first指向，子句链表头节点指向；同时在变元表内l对应变元的正（负）文字信息链表表头增加新增结点信息；增加子句数和文字数。

1. 删除CNF范式邻接链表中从表头开始第一个只含有文字l的单子句链表

status RemoveHeadClaus(Root \*r,int l);

参数l：(int)要删除的单子句的唯一文字值；

返回值：OK：删除成功

要删除的结点信息必在变元表内l对应变元的正（负）文字信息链表表头，移动删除free即可。此时还需要特判是否在子句链表第一个子句，若是第一个子句，则r->first=p->nextc;否则寻找从表头开始第一个只含文字l的单子句，再移动删除free即可。

1. 恢复认为文字l为真前的CNF邻接链表

status RecoverCNF(Root \*r,int l);

参数：(int)文字l:真值为1时求解出错

返回值：OK：邻接链表恢复成功

关键在于将原来DeleteClause和DelteLiteral的flag恢复为0即可，若l为正文字，对变元l1的正文字信息链表进行搜索，寻找被删除的子句，同时对变元l1的负文字信息链表进行搜索，寻找被删除的文字；若l为负文字，对变元l1的负文字信息链表进行搜索，寻找被删除的子句，同时对变元l1的正文字信息链表进行搜索，寻找被删除的文字。

1. 遍历CNF范式邻接链表

void ParadigmTrabverse(Root \*r);

依次遍历子句，进入后遍历子句链表节点输出文字值即可。

1. 保存CNF范式的解及求解时间信息

status SaveValue(ArgueValue \*ValueList,int solut,int time)

参数：(int)solut：CNF公式的解，有解为1，无解为0；

(int)time: DPLL求解时间；

返回值：ERROR：文件打开失败；

OK：解和时间存储成功

首先创建相同文件名但不同文件（.res格式）的文件，打开输入s solut（1为有解，0为无解），若solut为1则打印v后打印变元真值到文件中，否则不打印，最后打印t 和时间time；

1. status SAT(void);

SAT解CNF的流程控制函数；

1. 用第一种变元选取策略的递归算法DPLL函数

status DPLL1(int num,int op,int timesofDPLL);

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解

分两种情况：

1. 普通求解CNF文件：

初次进入DPLL函数，不进入单子句循环，直接采用第一种变元选取l=FindLiteral1，进而进入DPLL（l，1，2）；

若不是初次进入，则进入while（）循环进行单子句原则，将其变元表真值设置为1（l>0）或0（l<0）,同时IsInit设置为1（已经赋值）删除不成功则恢复原状态；搜索完所有单子句后退出再找新的文字值设置为真值，继续DPLL（l，1，2）；

若探测DPLL（l，1，2）不是OK，则设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1；若instacksize==0，则return FALSE; 因为l已是本次DPLL1函数第一个进行探测的变元，探测失败，恢复递归进入本次函数前的邻接链表状态。若探测成功，则有解return OK；

1. 生成数独终盘为2：

l一直为0；然后op=2进入随机赋值l，然后查找未被赋值的负文字链表的存在子句，找到未被删除的文字值，然后DPLL（l，1，2）进行求解，若探测DPLL（l，1，2）不是OK，则设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1；若instacksize==0，则return FALSE; 因为l已是本次DPLL1函数第一个进行探测的变元，探测失败，恢复递归进入本次函数前的邻接链表状态。若探测成功，则有解return OK；以便于创建9\*9随机数盘。

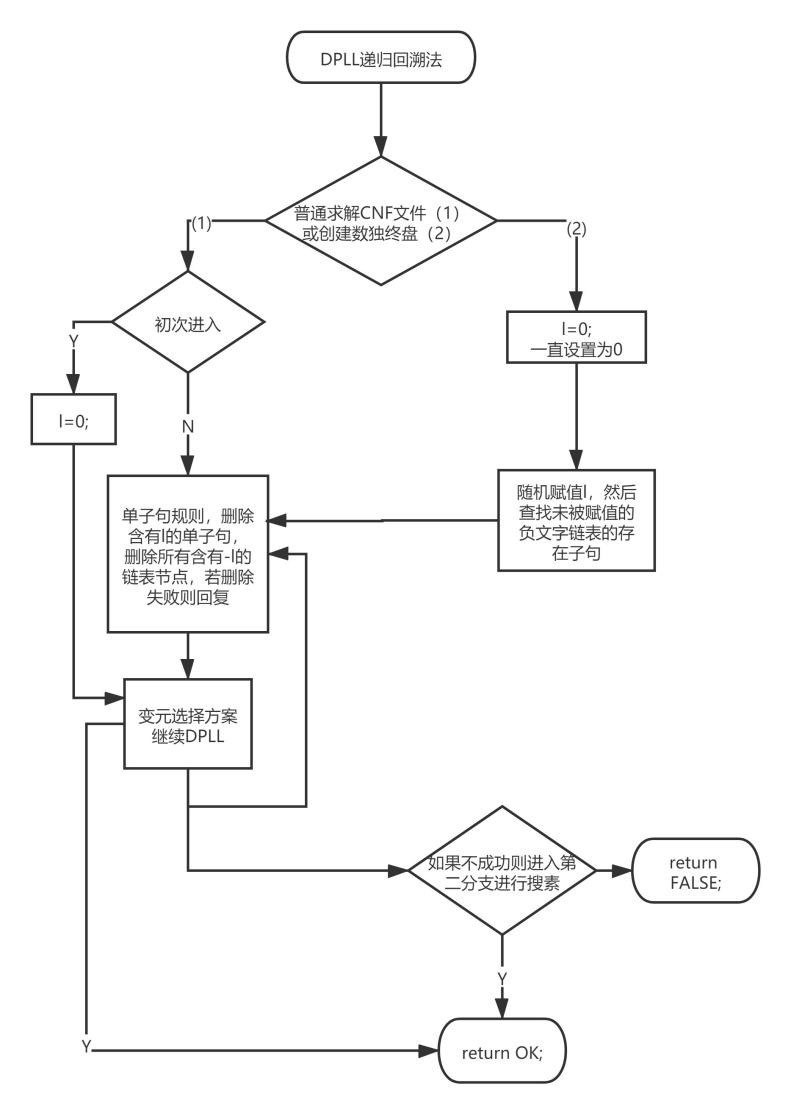


图3.2-3 递归算法DPLL函数

1. 用第二种变元选取策略的递归算法DPLL函数

status DPLL2(int num,int op,int timesofDPLL);

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解

第三，四种变元选取策略的递归算法DPLL函数与上同理；

1. 优化前版本DPLL函数

status DPLL\_Before(Root \*r,int op);

参数：(int)op：普通求解cnf文件为1，生成数独终盘为2；

返回值：FALSE：公式无解；

TRUE：公式有解

先建立Memory栈保存每次DPLL循环过程中真值设为1的文字，并且初始化为0，然后单子句策略让c赋值为找到的文字对应的子句链表节点，循环删除含有对应的文字值的子句和删除所有含有对应文字值的相反数的子句链表节点，删除失败则回复，搜索完现状所有单子句后就找新的词，添加单子句然后继续搜索；l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1。

1. 将未赋真值的变元（求解过程中化简舍去）赋真值

void AnswerComplete(void)；

while（）循环赋值；

1. 检查SAT求解结果正确性

status AnswerCheck(int solut)；

参数：(int)solut：公式求解结果，有解为1，无解为0；

返回值：TRUE：求解正确；

FALSE：求解错误

如果solut为1，则判断每个子句中若有一个节点文字值真值为1，则说明子句为真，否则子句中无真值为1的文字，子句真值为0，求解错误；  
 如果solut为0，子句中有文字真值为1，子句真值为1，所有子句节点真值应为0，子句真值为0，求解正确；

#### 3.2.3 Sudoku算法处理

1. status Sudoku(void);

Sudoku的流程控制台；

1. 创建数独问题转化为SAT问题后的cnf文件

FILE \* CreateSudokuFile(void)；

先创建.cnf文件，初始化填入共有729个变元，9\*9个数独空格每个格对应9个变元，填入1～9中某一值则对应变元为真，其他为假，共10287个子句；

三个约束：每行、每列不重复，3\*3小格内部不重复；（根据条件代码实现）

同时每个小格可以填入1～9任何值。

1. 创建新的随机9\*9数独终盘

status CreateSudoku(void)

初始化棋盘sudoku\_table，创建基础CNF文件（调用CreateSudokuFile(void)，再创建CNF范式邻接表和变元真值表，开始随机数赋值，赋值后删除含有文字值的子句和所有含有文字值相反数的子句节点，再DPLL查看是否有解。

1. 将数独txt文件转化为cnf

FILE \* SetCNFfile(void);

读入棋盘的转化求解的辅助函数；

1. 求解数独cnf文件并转化为数独终盘形式输出

status SolveSudoku(void);

读入棋盘文件（.txt格式），转化为.cnf格式进行DPLL求解，再将解储存下来一一赋值到sudoku\_table数组内；

1. 对已知数独终盘挖num个洞生成有唯一解的数独问题

status DigHole(int num);

参数：(int)num：挖洞的数目

先初始化要挖洞的棋盘（int table\_diged[9][9]）和记录挖洞的数组（int diged\_cells[82]），创建基础CNF文件和棋盘，读入已知数独终盘创建最终CNF文件，复制sudoku\_table的值至数组table\_diged，开始挖洞；

挖第一个洞有随机性，必定有唯一的解，将CNF公式邻接链表恢复至未知第一个洞位置的填入值，即其对应变元真值时的状态，再销毁即可；

挖后续的洞，循环创建基础CNF文件和棋盘，读入修改后数独终盘创建CNF文件，再随机挖洞。之后DPLL求解是否有唯一解，有唯一解才可以继续挖下一个洞。最后输出生成棋盘。

1. 完善数独终盘（该方法也是求数独终盘的方法）

status SudokuComplete(void);

根据创建的随机数独盘进行求解，横向竖向嵌套循环扫描得到数独终盘。

1. 数独对应SAT变元表转化为二维数组

status CNFSudokuTableTransform(void);

根据SAT变元表转化为二维数组即可。

1. 输出数独盘

status SudokuTablePrint(void);

嵌套循环输出数度盘即可，为了更显得简洁明了加了边框进行修饰。

**4系统实现与测试**

**4.1系统实现**

#### 4.1.1软硬件环境

1. 硬件环境：

处理器：Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.80GHz

机带RAM：8.00GB

系统类型：64位操作系统，基于x64的处理器

1. 软件环境：

Windows10下Dev-C++ 5.11

#### 4.1.2数据类型定义

1. 宏定义

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define INCREASEMENT 100

1. 数据类型定义

typedef int status;

1. 全局变量

char filename[1000]; //文件名

long instacksize;//记录栈stack的大小

ArgueValue \*ValueList;//变元真值表

Root \*r;//链表

int \*Frequency;//变元出现次数辅助数组

int \*stack;//DPLL辅助栈

int sudoku\_table[9][9];//定义全局变量int类型二维数组存储数独盘

int users\_sudoku[9][9];//存储输出的含空格数独问题格局

1. 存储结构

/\*定义子句链表结点结构类型\*/

typedef struct Clause{

int literal;//记录子句中的文字

int flag;

//标记该文字是否已被删除，未删除时值为0，否则值为使之删除的变元序号便于回复

struct Clause \*nextl;//指向该子句中下一文字的指针

struct Clause \*litline;//指向整个CNF公式邻接链表中下一个文字相同的子句结点，用于回复与方便查找

}Clause;

/\*定义CNF范式链表结点（即子句链表头结点）结构类型\*/

typedef struct Paradigm{

int number;//子句中显示的文字数，便于找单子句

int flag;//标记该子句是否已被删除，未删除时值为0，否则值为使之删除的变元序号便于回复

struct Paradigm \*nextc;//指向下一子句的头结点

struct Clause \*sentence;//子句头指针

}Paradigm;

/\*定义CNF范式链表头结点类型，存储CNF范式信息\*/

typedef struct Root{

int litsize;//存储文字数量

int parasize;//存储子句数量

Paradigm \*first;//指向第一个子句

}Root;

/\*Dpll会用到的两个结构，用于更快寻找正文字和负文字 \*/

/\*定义指向子句链表头结点的指针链表结点结构类型\*/

typedef struct Paraline{

Paradigm \*claline;//指向子句链表头结点Paradigm

struct Paraline \*next;//指向下一链表结点

} Paraline;

/\*定义文字相关信息链表结构类型\*/

typedef struct LitTrabverse{

Paraline \*Tra\_cla;//指向含有该正文字或负文字的子句头结点链表的头结点

Clause \*Tra\_lit;//指向该正文字或负文字的文字结点

}LitTrabverse;

/\*定义求解数独文件时所需的存储变元信息结构类型\*/

typedef struct sudokusolver{

int x;//存储行信息

int y;//存储列信息

int z;//存储变元对应1～9数值信息

} sudokusolver;

/\*定义存储变元信息的变元线性表结点结构类型\*/

typedef struct ArgueValue{

int Value;//变元的真值

int IsInit;//变元是否已赋值

int Occur\_times;//变元在所有子句中出现的总次数，用于分裂政策的选择

LitTrabverse Pos;//变元所有正文字的相关信息结构

LitTrabverse Neg;//变元所有负文字的相关信息结构

sudokusolver xyz;//求解数独文件时所需的变元信息

} ArgueValue;

#### 4.1.3函数声明以及函数调用关系

1. CNF文件处理

/\*创建CNF范式邻接链表及变元表\*/

status CreateParadigm(FILE \*\*fp);

/\*创建子句链表及文字链表\*/

int CreateClause(FILE \*\*fp,Clause \*\*sentence,Paradigm \*ClausHead,int first);

/ \*销毁所有链表及线性表结构\*/

status DestroyParadigm(Root \*r);

/\*判断CNF范式中是否还含有单子句\*/

int HasUnitClause(Root \*r);

/\*判断CNF范式中是否还含有单子句（优化前版本）\*/

Clause \* HasUnitClause\_Before(Root \*r);

/\*判断指针c指向的子句链表是否为单子句链表\*/

status isUnitClause(Paradigm \*c);

/\*遍历CNF范式邻接链表\*/

void ParadigmTrabverse(Root \*r);

/\*删除出现了文字l的所有单子句\*/

status DeleteClause(Root \*r,int l);

/\*在CNF范式邻接链表表头添加只含有文字l的单子句链表\*/

status AddClause(Root \*r,int l);

/\*删除CNF范式邻接链表中从表头开始第一个只含有文字l的单子句链表\*/

status RemoveHeadClaus(Root \*r,int l);

/\*删除所有文字为-l的子句链表结点\*/

status DeleteLiteral(Root \*r,int l);

/\*恢复认为文字l为真时对CNF范式邻接链表所作的操作\*/

status RecoverCNF(Root \*r,int l);

/\*保存CNF范式的解及求解时间信息\*/

status SaveValue(ArgueValue \*ValueList,int solut,int time);

1. DPLL算法

/\*在整个CNF公式中取一个文字\*/

/\*取每次DPLL处理后公式中Occur\_Times最大的文字\*/

int FindLiteral1(Root \*r);

/\*取原公式中Occur\_Times最大的文字\*/

int FindLiteral2(Root \*r);

/\*在整个CNF公式中取一个文字（优化前版本）\*/

Clause \* FindLiteral\_Before(Root \*r);

/\*DPLL流程控制\*/

status SAT(void);

/\*采用第一种变元选取策略的递归算法DPLL函数\*/

status DPLL1(int num,int op,int times);

/\*采用第二种变元选取策略的递归算法DPLL函数\*/

status DPLL2(int num,int op,int times);

/\*优化前版本DPLL\*/

status DPLL\_Before(Root \*r,int op);

/\*完善SAT求解结果\*/

void AnswerComplete(void);

/\*检查SAT求解结果正确性\*/

status AnswerCheck(int solut);

函数功能联合：读取CNF文件，创建对应变元真值表和CNF范式链表后，选取任意一种选择变元方法，进入对应DPLL的函数进行求解。

1. Sudoku

/\*数独流程控制\*/

status Sudoku(void);

/\*创建数独问题转化为SAT问题后的cnf文件\*/

FILE \* CreateSudokuFile(void);

/\*创建新的随机9\*9数独终盘\*/

status CreateSudoku(void);

/\*求解数独cnf文件并转化为数独终盘形式输出\*/

status SolveSudoku(void);

/\*对已知数独终盘挖num个洞生成有唯一解的数独问题\*/

status DigHole(int num);

/\*完善数独终盘\*/

status SudokuComplete(void);

/\*将数独txt文件转化为cnf \*/

FILE \* SetCNFfile(void);

/\*数独对应SAT变元表转化为二维数组\*/

status CNFSudokuTableTransform(void);

/\*输出数独盘\*/

status SudokuTablePrint(void);

函数功能联合：

1. 读取棋盘.txt文件，转化为CNF文件，再创建对应变元真值表和CNF范式链表，棋盘数据刷新CNF文件后进行DPLL求解，再根据变元真值表信息转化为二维数组棋盘遍历打印；
2. 随机数生成棋盘，进行DPLL求解能得到数据终盘后挖洞，之后DPLL求解，能有唯一解才确定为数独初始盘，之后赋值到二维数组遍历打印（可交互）；

**4.2系统测试**

#### 4.2.1交互系统展示

主交互界面，SAT求解界面，数独游戏界面展示，如图4.2-1，4.2-2，4.2-3所示

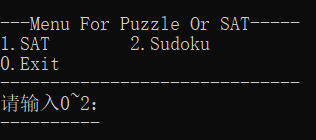


图4.2-1 主交互界面

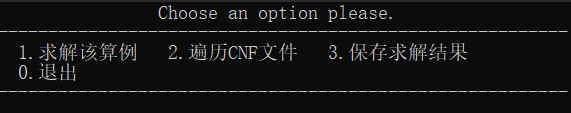


图4.2-2 SAT求解界面

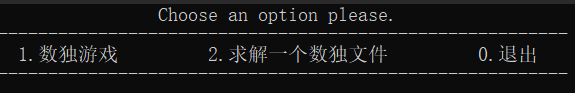


图4.2-3 数独游戏界面

#### 4.2.2 CNF文件处理模块测试和DPLL算法模块测试

首先简介下过程：在交互主界面点击1，并输入想要求解的CNF文件名；进入SAT求解界面。选择求解该算例，有优化1，优化2，优化前三个选项，求解出来时会打印出真值；求解完成后可以选择遍历或者保存结果到.res文件中；

1. 测试算例1：SAT测试备选算例\其它可供选择使用的算例\problem1.cnf，先读入所要求解的CNF文件名（如图4.2.2-1所示），遍历结果（如图4.2.2-2所示）测试结果如图4.2.2-3，图4.2.2-4,图4.2.2-5所示：

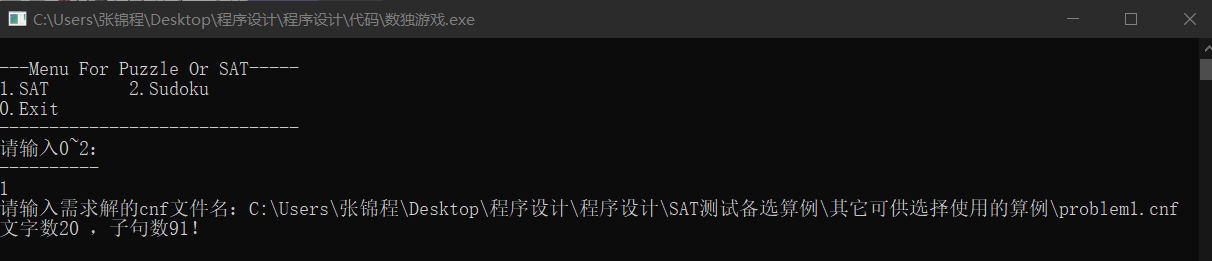


图4.2.2-1 读入所要求解的CNF文件名

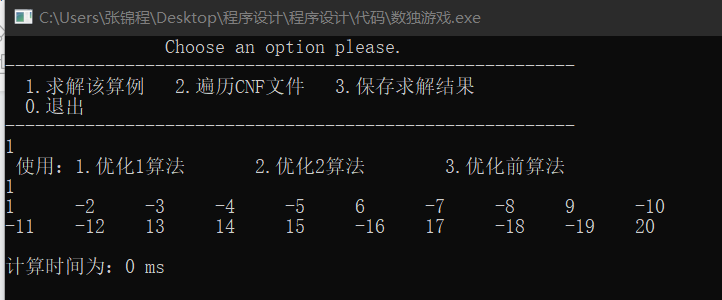


图4.2.2-2 优化1的SAT求解

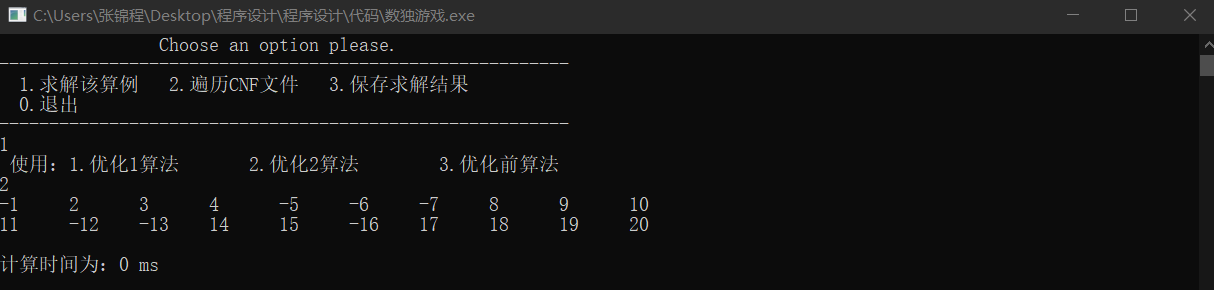


图4.2.2-3 优化2的SAT求解

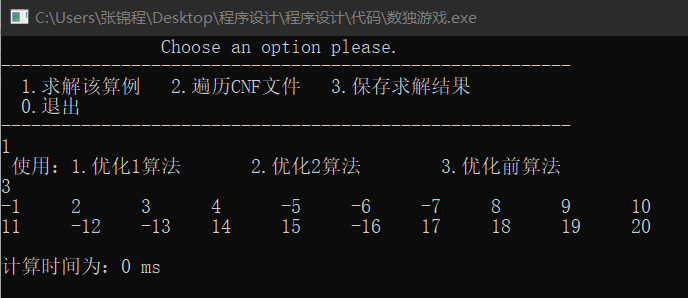


图4.2.2-4 优化前SAT求解

优化率（1）：0。

优化率（2）：0。

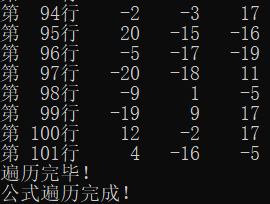
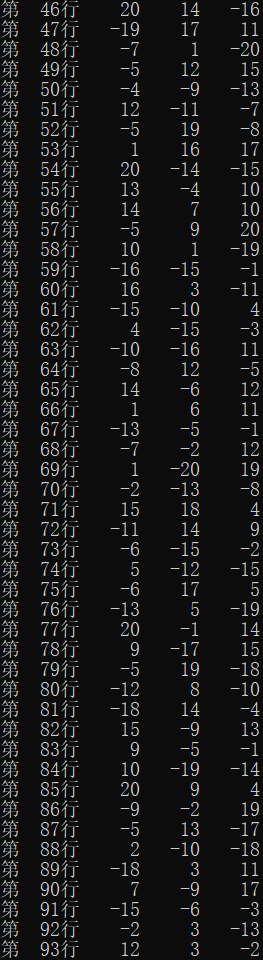
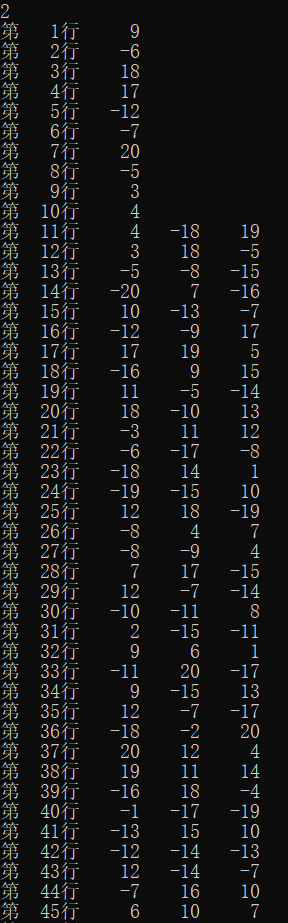


图4.2.2-5 遍历结果

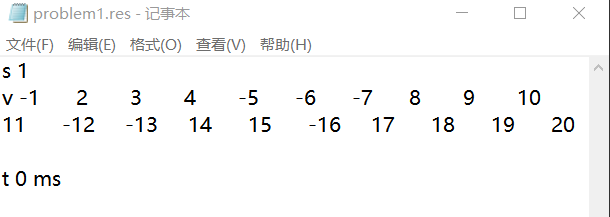


图4.2.2-6 保存结果

提示：接下来读入文件，遍历结果将省略。

1. 测试算例2：SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v10\_c100.cnf，如图所示。

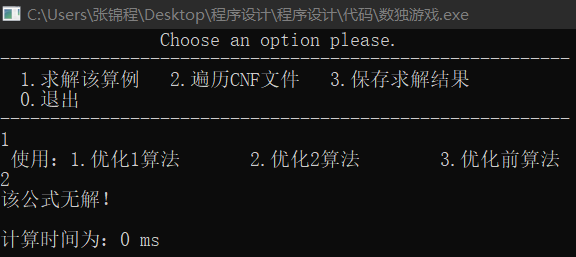
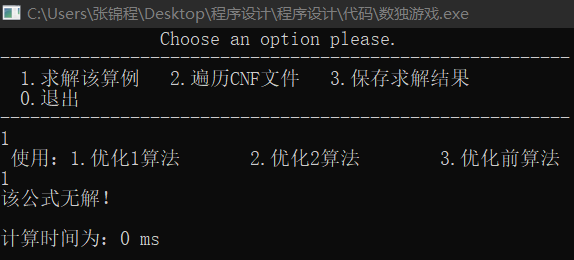


图4.2.2-7 优化1的SAT求解 图4.2.2-8 优化2的SAT求解

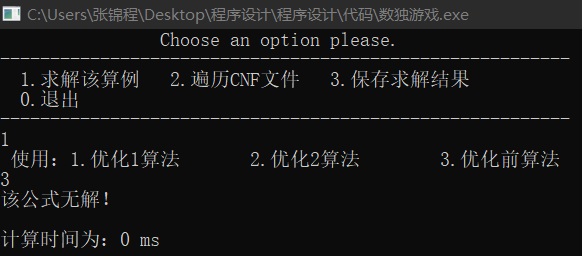


图4.2.2-8 优化前SAT求解

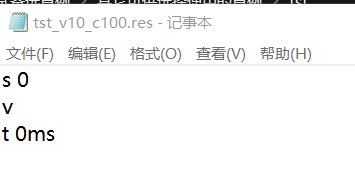


图4.2.2-9 保存结果

优化率（1）：0。

优化率（2）：0。

提示：接下来保存结果的文件将省略。

1. 测试算例3：SAT测试备选算例\其它可供选择使用的算例\tst\flat30-1.cnf，先读入所要求解的CNF文件名（如图4.2.2-1所示），测试结果如图4.2.2-2，图4.2.2-3,图4.2.2-4所示：

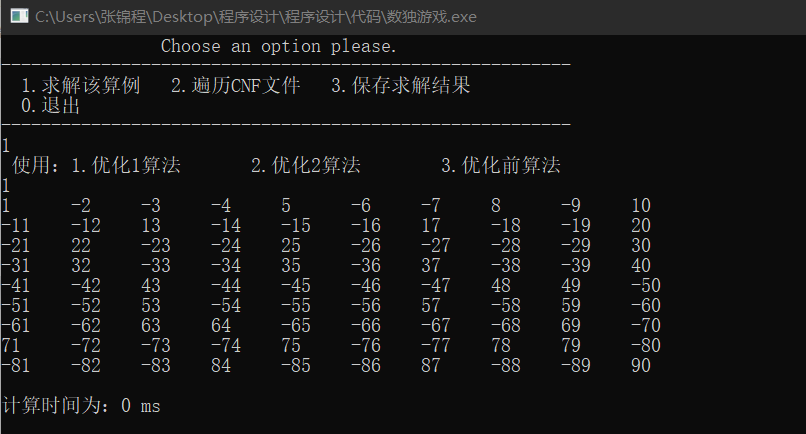


图4.2.2-10 优化1的SAT求解

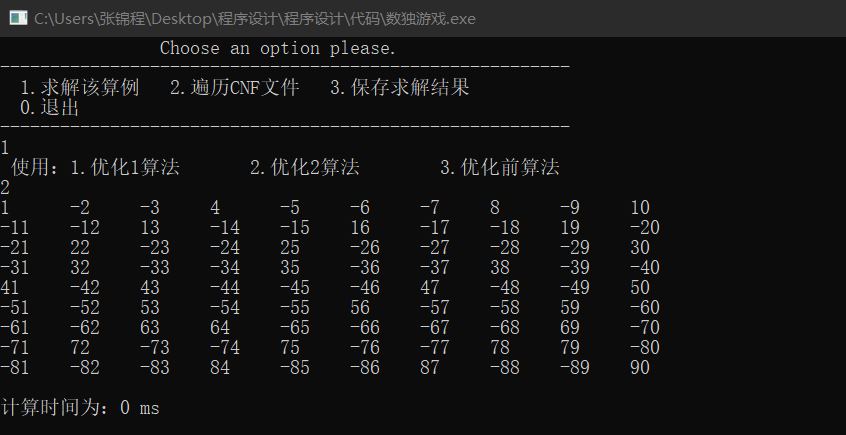


图4.2.2-11 优化2的SAT求解

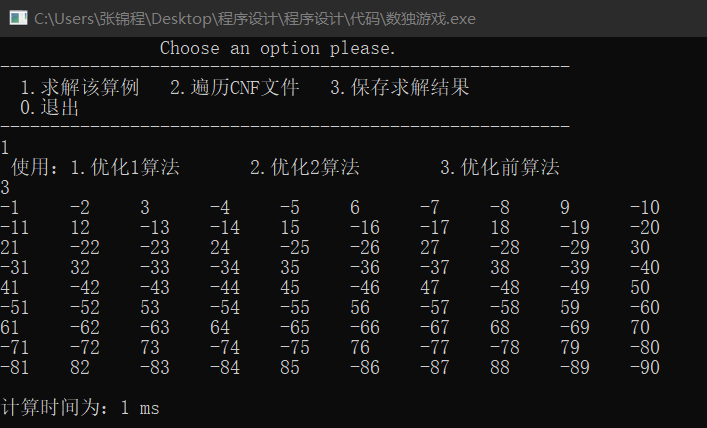


图4.2.2-12 优化前SAT求解

优化率（1）：[(1-0)/1]\*100%=100%

优化率（2）：[(1-0)/1]\*100%=100%

1. 测试算例4：SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v100\_c400.cnf,如图所示：

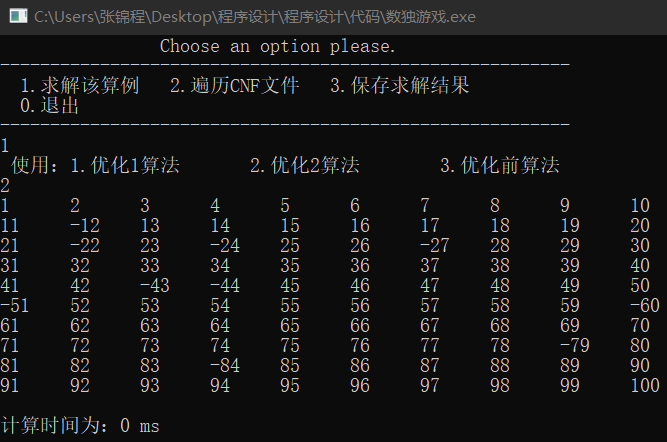


图4.2.2-13 优化1的SAT求解 图4.2.2-14 优化2的SAT求解

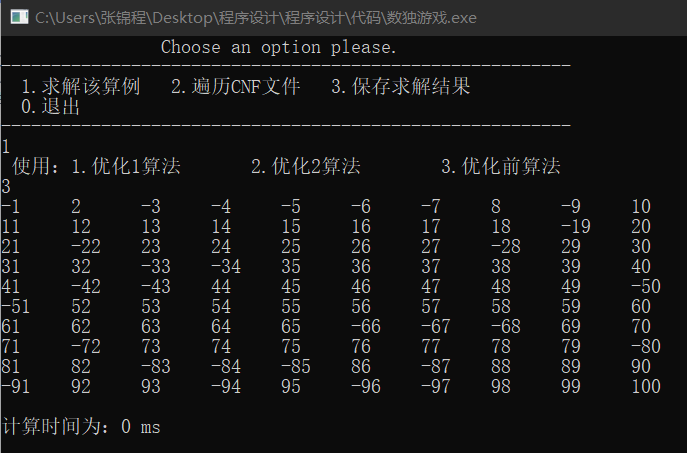


图4.2.2-15 优化前的SAT求解

优化率（1）=0

优化率（2）=0

1. 测试算例5：SAT测试备选算例\其它可供选择使用的算例\tst\flat30-99.cnf

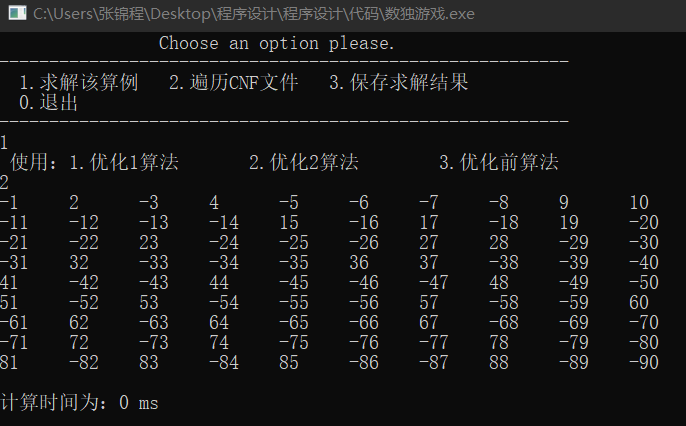
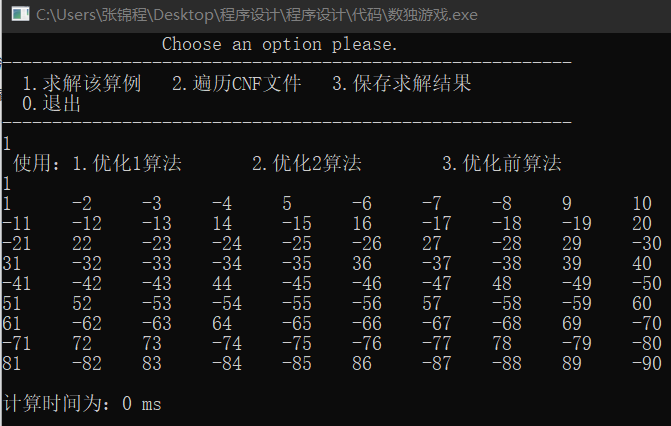


图4.2.2-16 优化1的SAT求解 图4.2.2-17 优化2的SAT求解

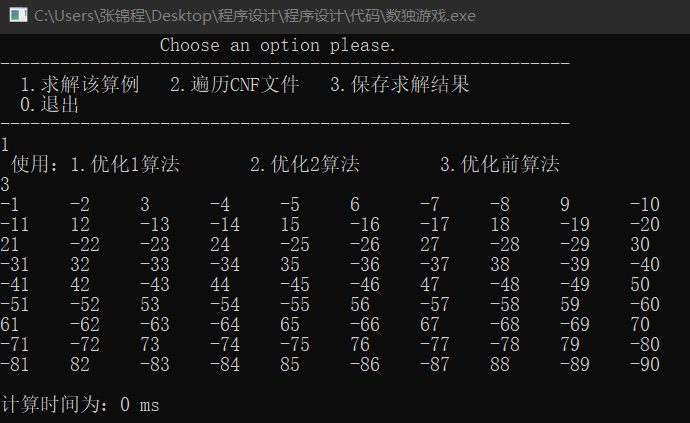
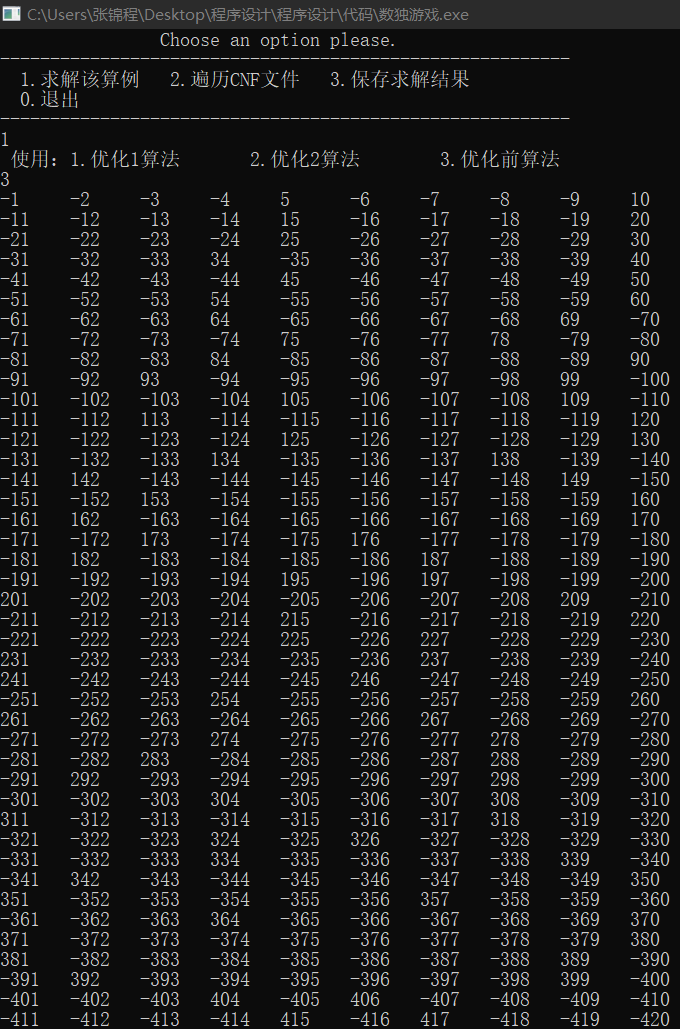


图4.2.2-18 优化前的SAT求解

优化率（1）：0

优化率（2）：0

1. 测试算例6：SAT测试备选算例\其它可供选择使用的算例\tst\sw100-1.cnf，如图所示：



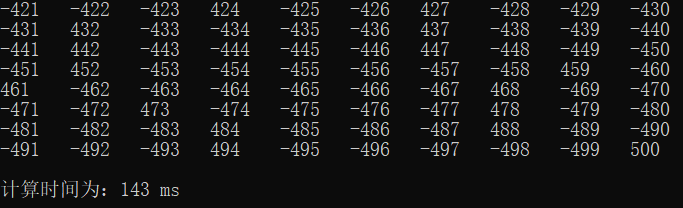


图4.2.2-19 优化前的SAT求解

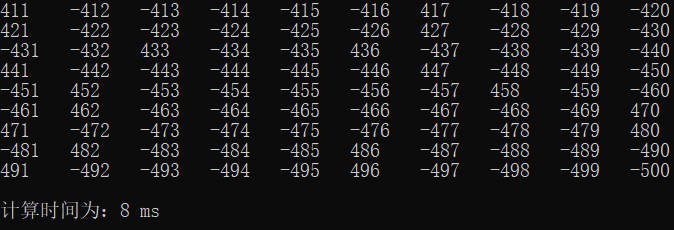
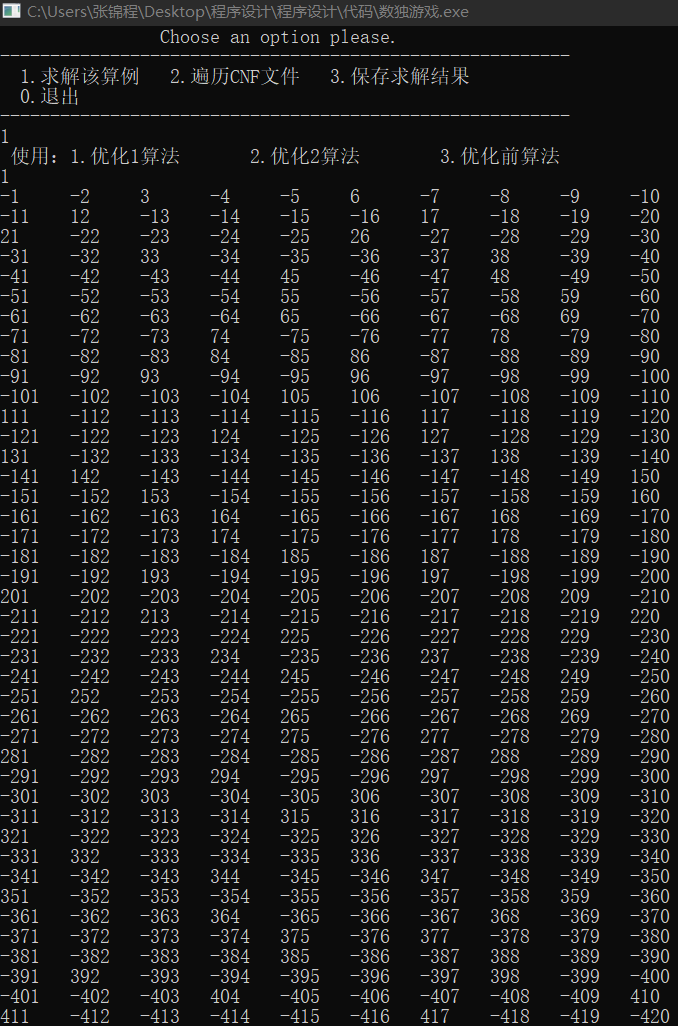


图4.2.2-20 优化1的SAT求解

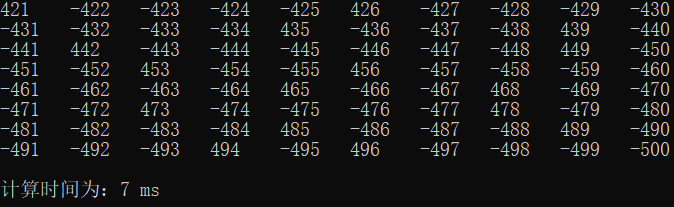
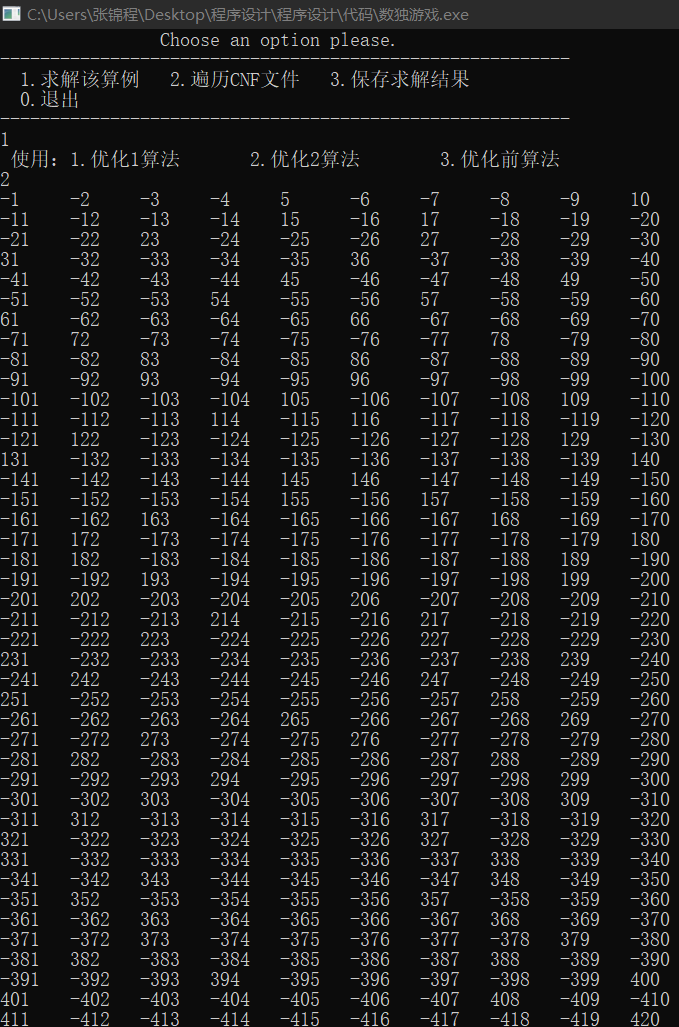


图4.2.2-21 优化2的SAT求解

优化率（1）：[(143-8)/143]\*100%=94.4%

优化率（2）：[(143-7)/143]\*100%=95.1%

1. 测试算例7：SAT测试备选算例\其它可供选择使用的算例\ais\ais12.cnf，如图所示：

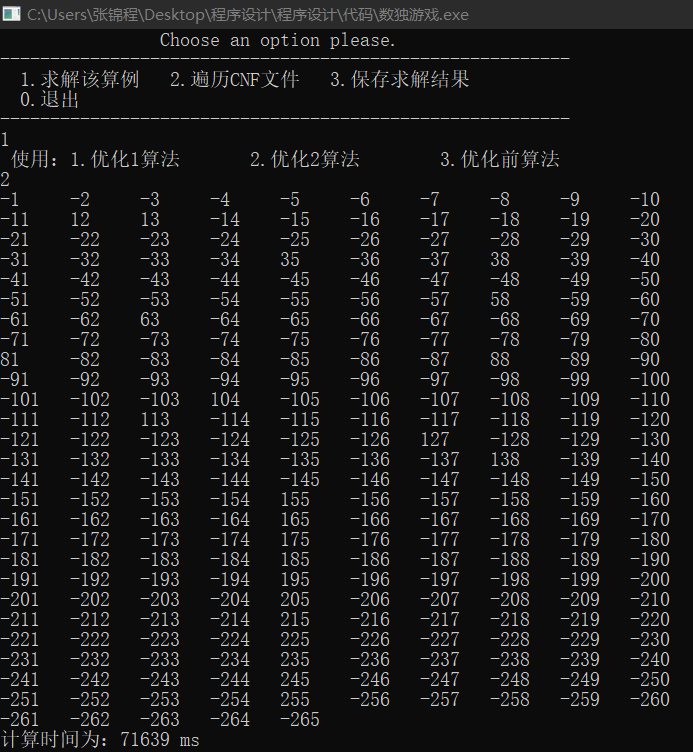
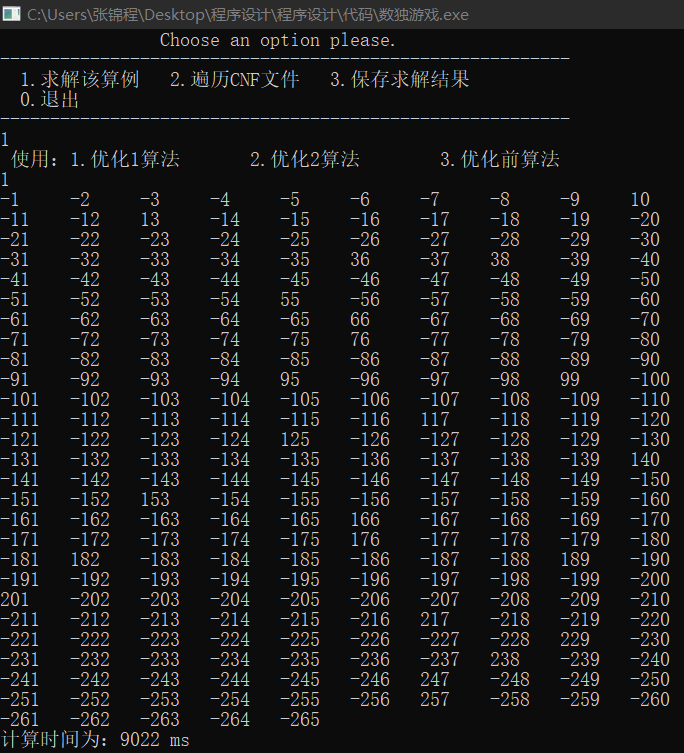


图4.2.2-22 优化1的SAT求解 图4.2.2-23 优化2的SAT求解



图4.2.2-24 优化前的SAT求解

优化率（1）：[(114846-9022)/114846]\*100%=92.1%

优化率（2）：[(114846-71639)/114846]\*100%=37.6%

1. 测试算例8：SAT测试备选算例\满足算例\M\sud00001.cnf,如图所示:

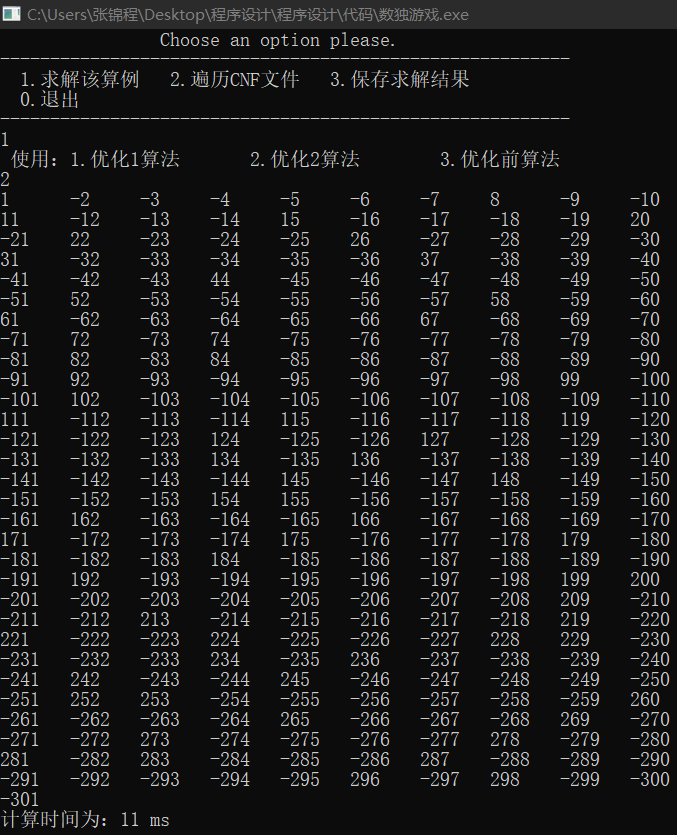
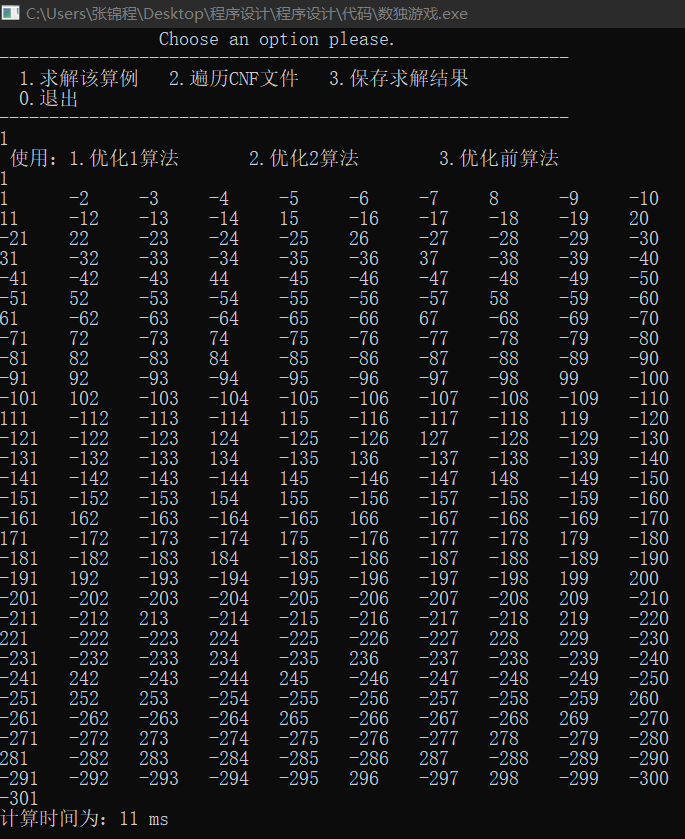


图4.2.2-25 优化1的SAT求解 图4.2.2-26 优化2的SAT求解

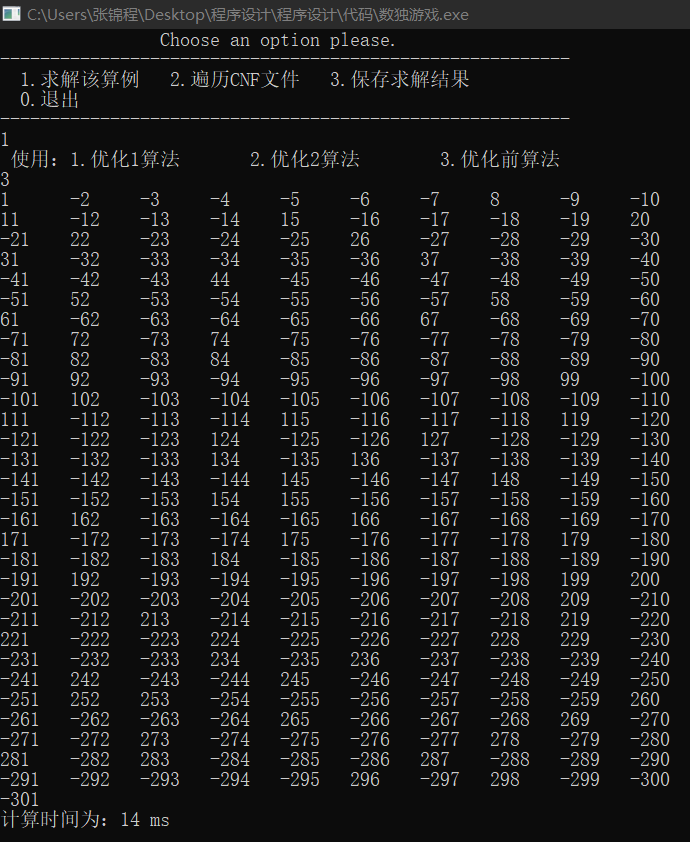


图4.2.2-27 优化前的SAT求解

优化率（1）：[(14-11)/14]\*100%=21.4%

优化率（2）：[(14-11)/14]\*100%=21.4%

1. 测试算例9：SAT测试备选算例\满足算例\M\sud00012.cnf,如图所示:

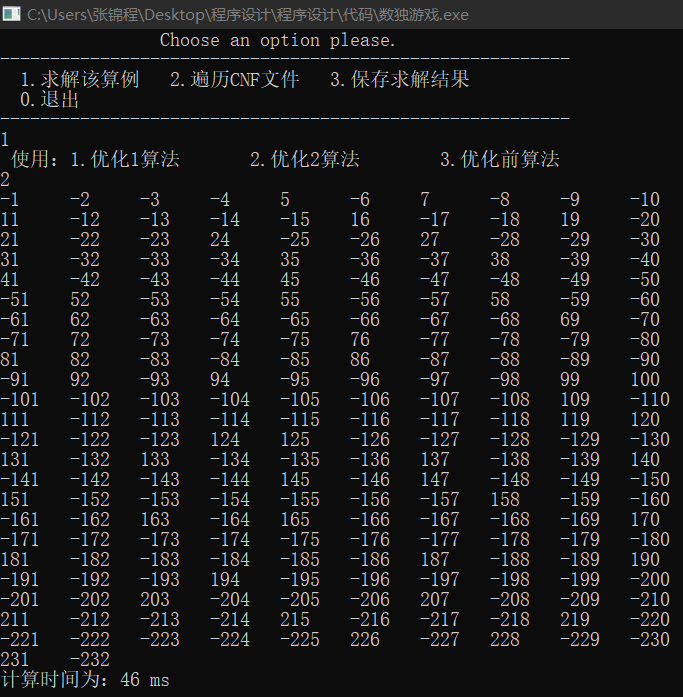
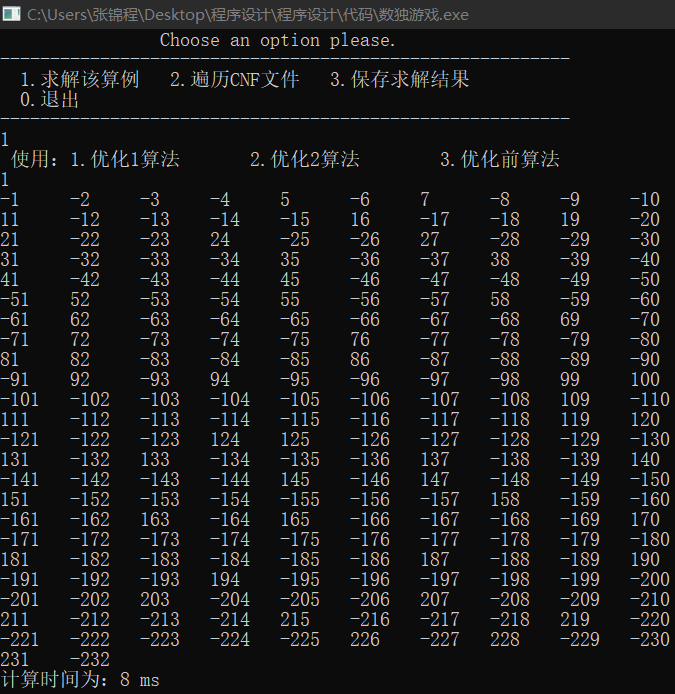


图4.2.2-28 优化1的SAT求解 图4.2.2-29 优化2的SAT求解

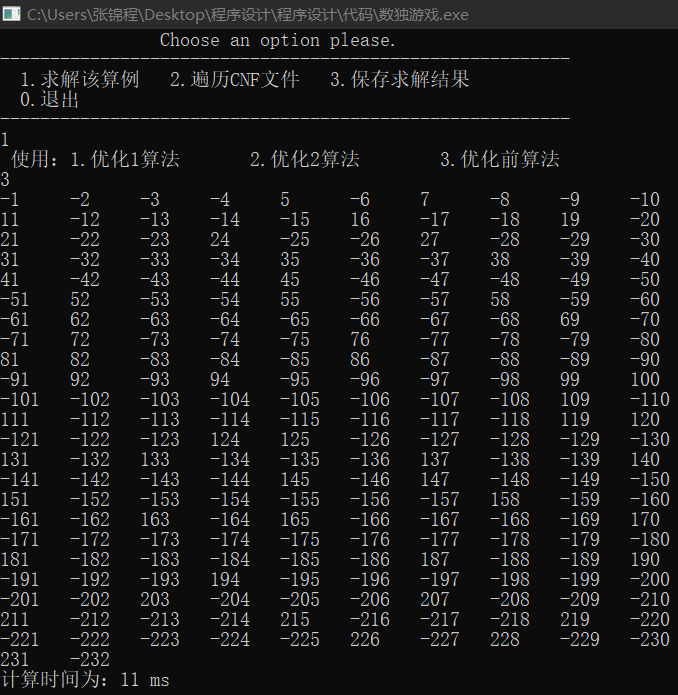


图4.2.2-30 优化前的SAT求解

优化率（1）：[(11-8)/11]\*100%=27.3%

优化率（2）：[(11-46)/11]\*100%=-318.2%

1. 测试算例10：SAT测试备选算例\满足算例\L\sssc.cnf，如图所示：

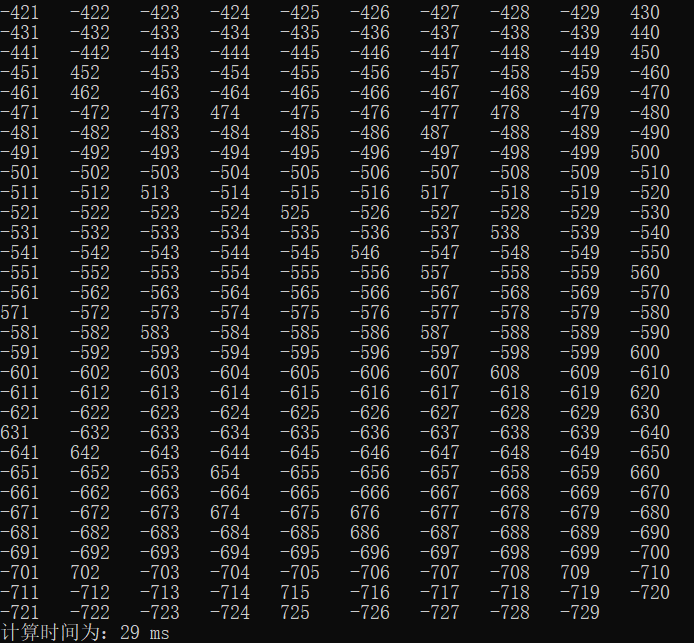
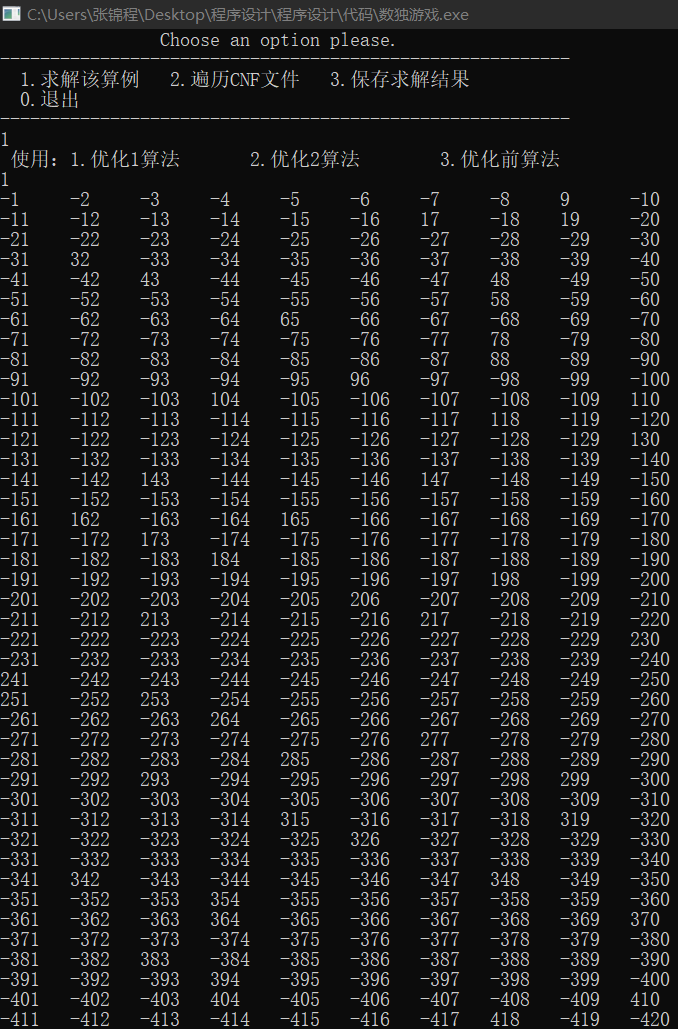


图4.2.2-31 优化1的SAT求解

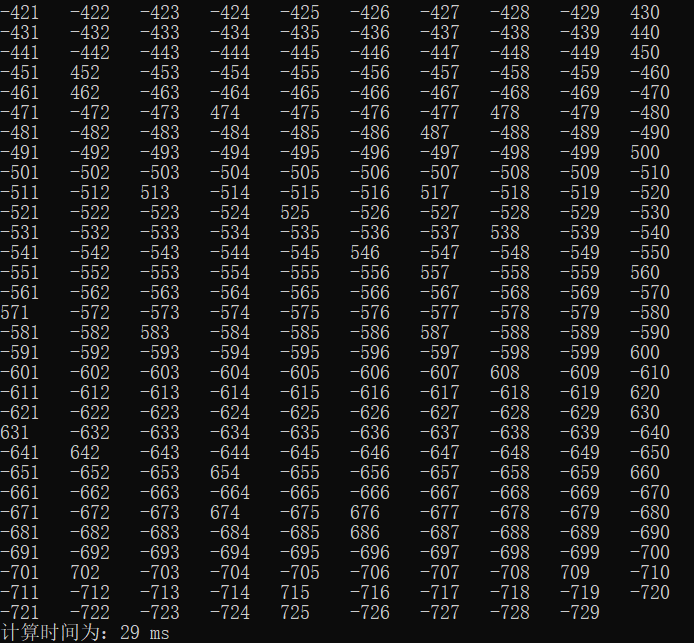
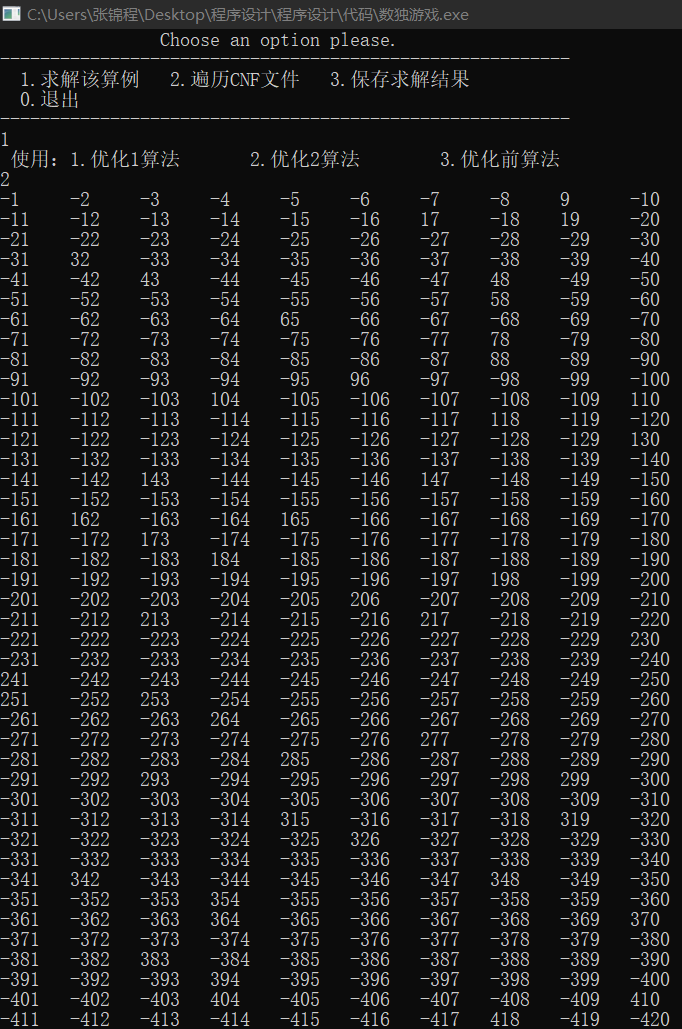


图4.2.2-32 优化2的SAT求解

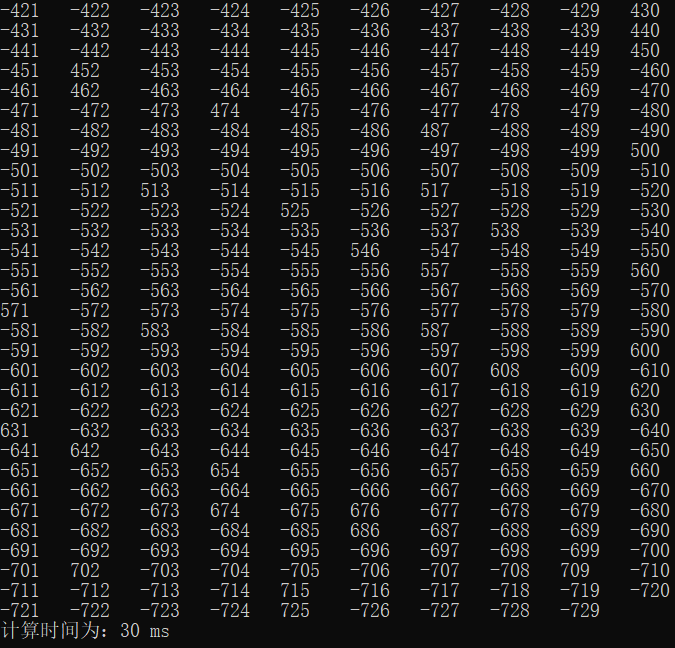
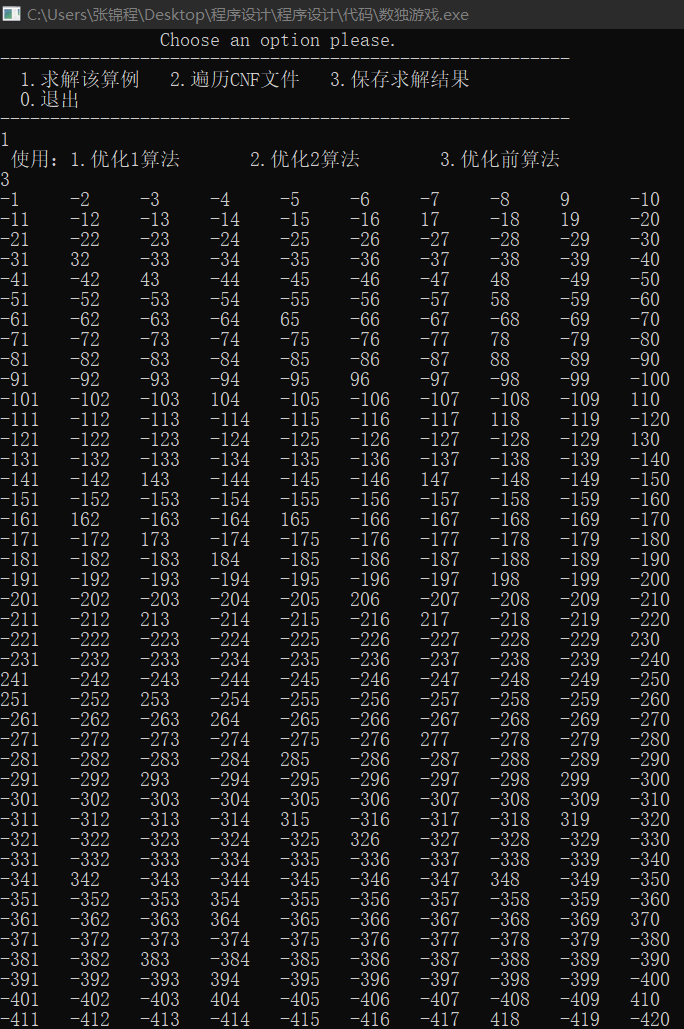


图4.2.2-33 优化前的SAT求解

优化率（1）：[(30-29)/30]\*100%=3.3%

优化率（2）：[(30-29)/30]\*100%=3.3%

1. 测试算例11：SAT测试备选算例\其它可供选择使用的算例\tst\qg4-08.cnf，如图所示：

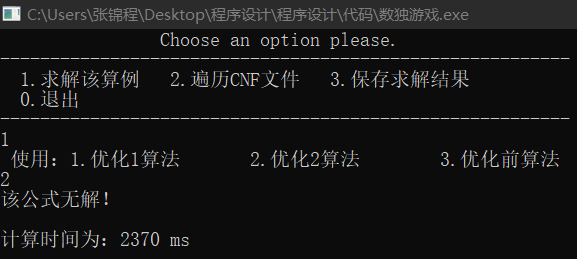
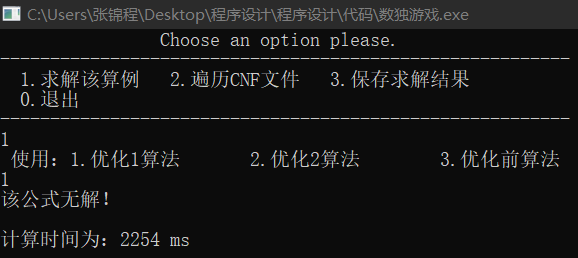


图4.2.2-34 优化1的SAT求解 图4.2.2-35 优化2的SAT求解

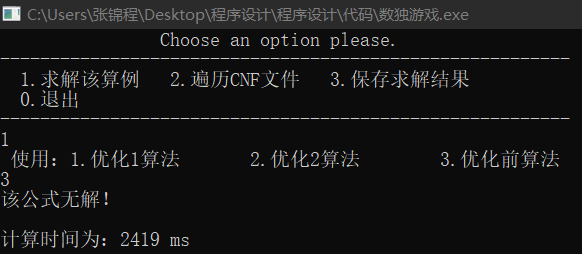


图4.2.2-36 优化前的SAT求解

优化率（1）：[(2419-2254)/2419]\*100%=6.8%

优化率（2）：[(2419-2370)/2419]\*100%=2.0%

1. 测试算例12：SAT测试备选算例\其它可供选择使用的算例\tst\bw\_large.a.cnf，如图所示：

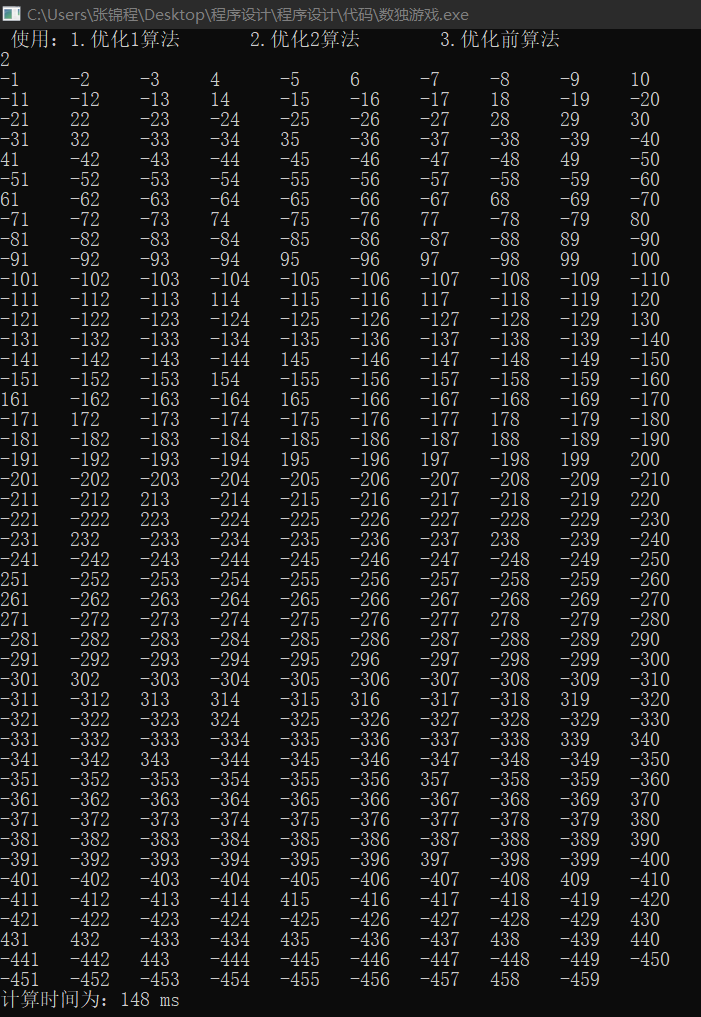
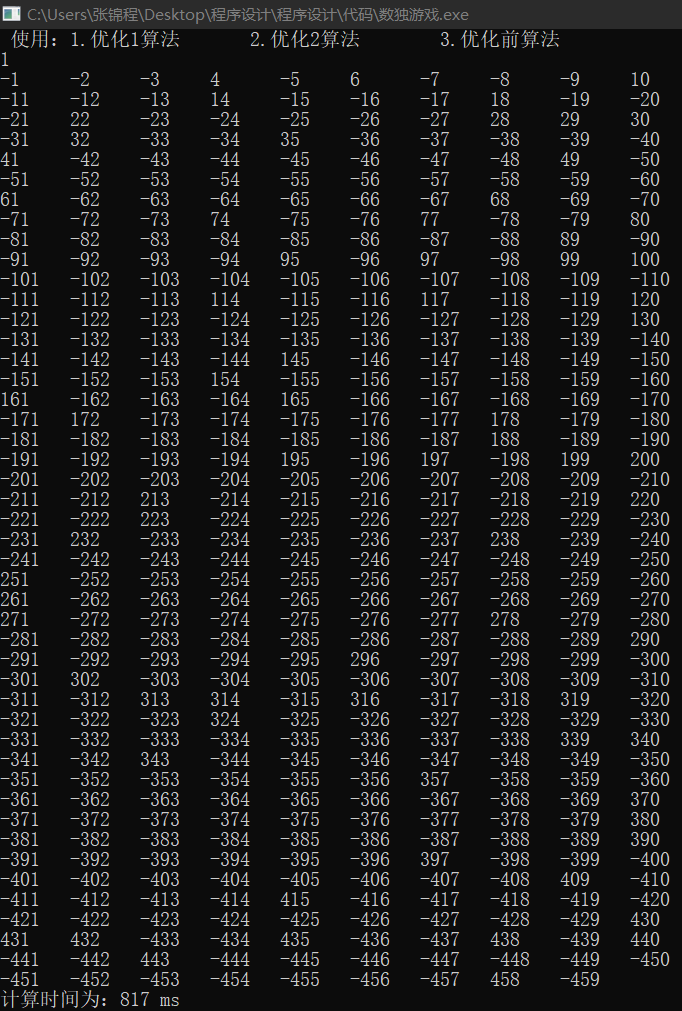


图4.2.2-37 优化1的SAT求解 图4.2.2-38 优化2的SAT求解

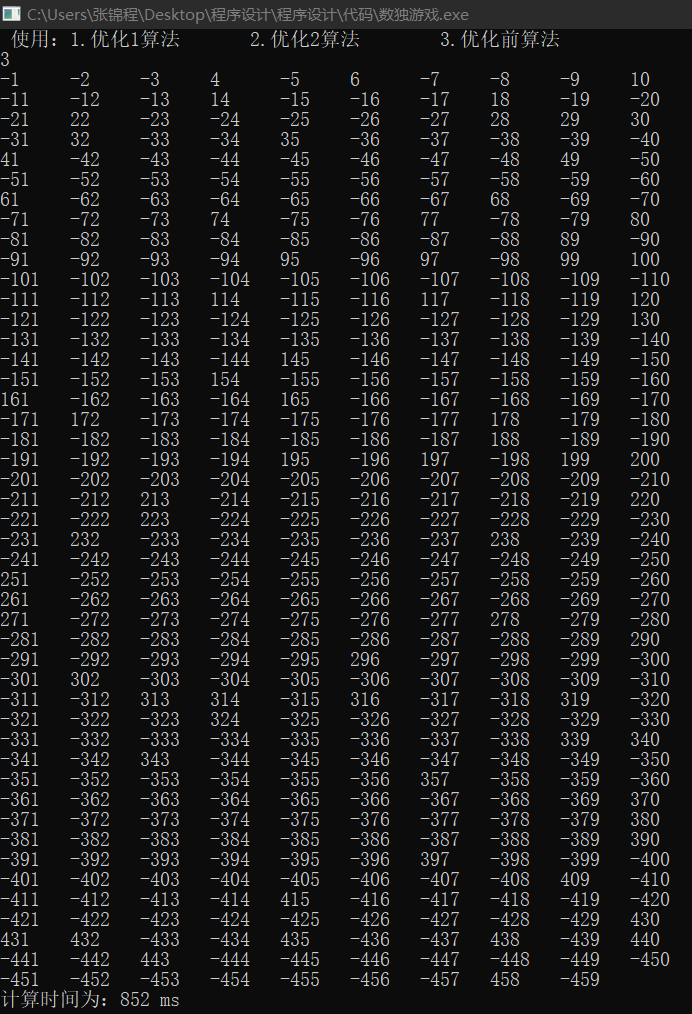


图4.2.2-39 优化前的SAT求解

优化率（1）：[(852-817)/852]\*100%=4.1%

优化率（2）：[(852-1135)/852]\*100%=-33.2%

1. 测试算例13：SAT测试备选算例\满足算例\M\ssm.cnf,如图所示：

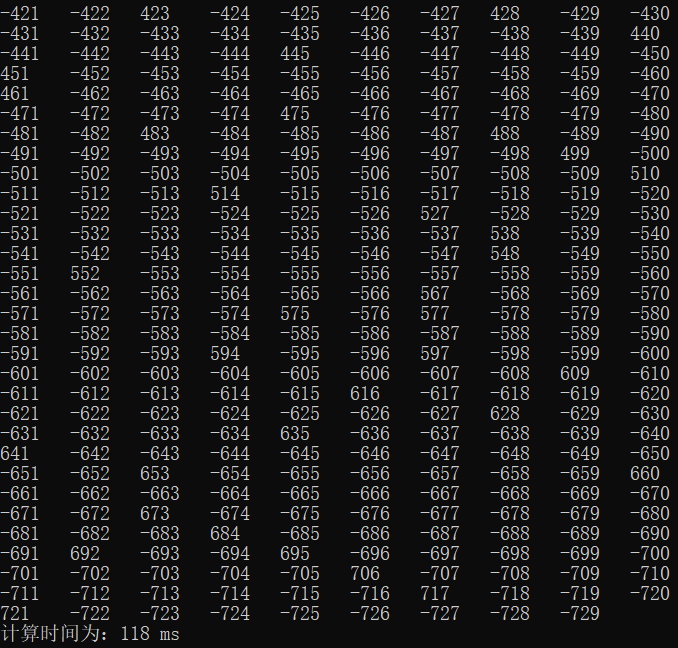
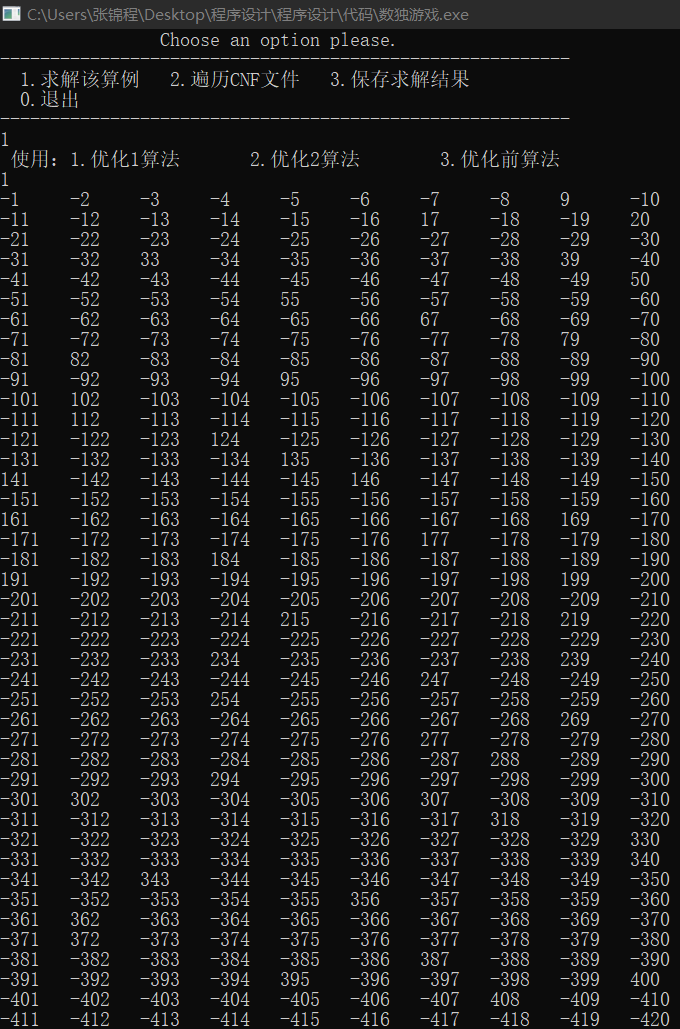


图4.2.2-40 优化1的SAT求解

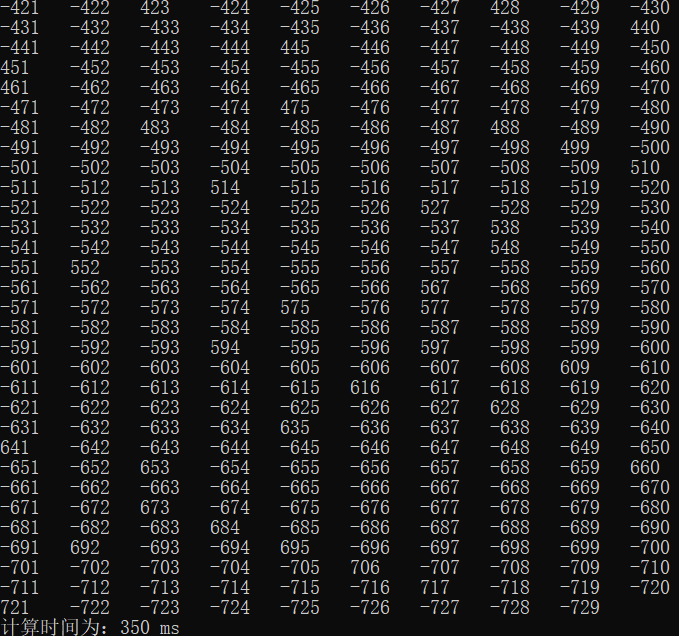
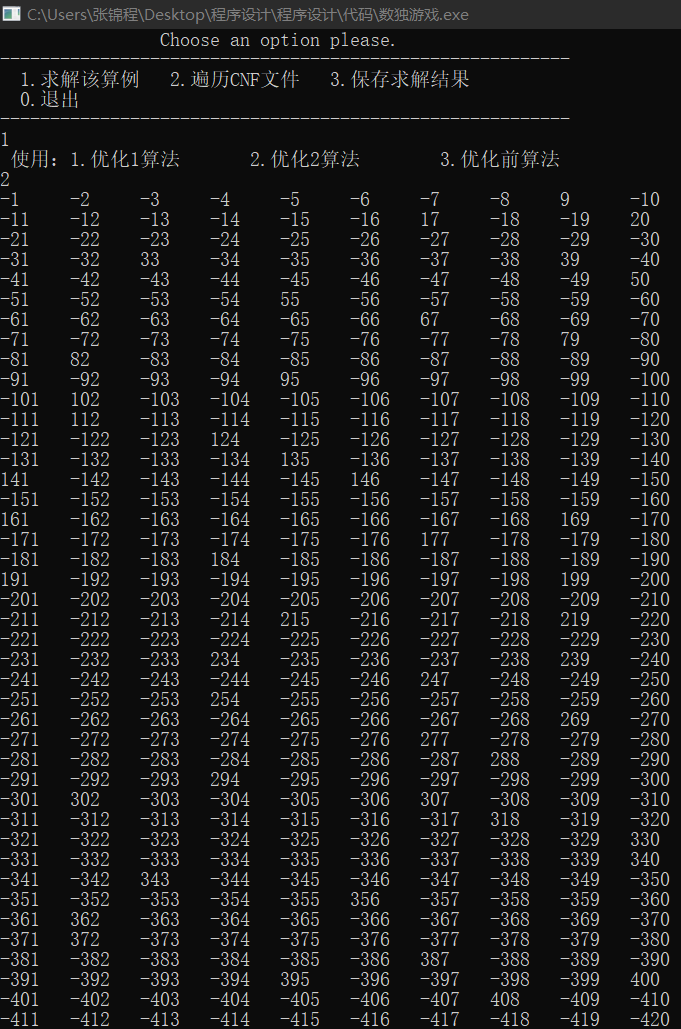


图4.2.2-41 优化2的SAT求解

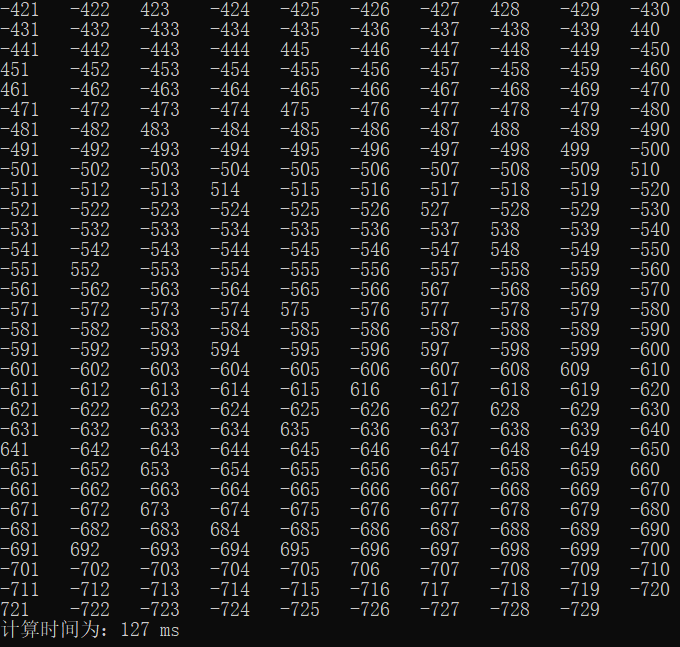
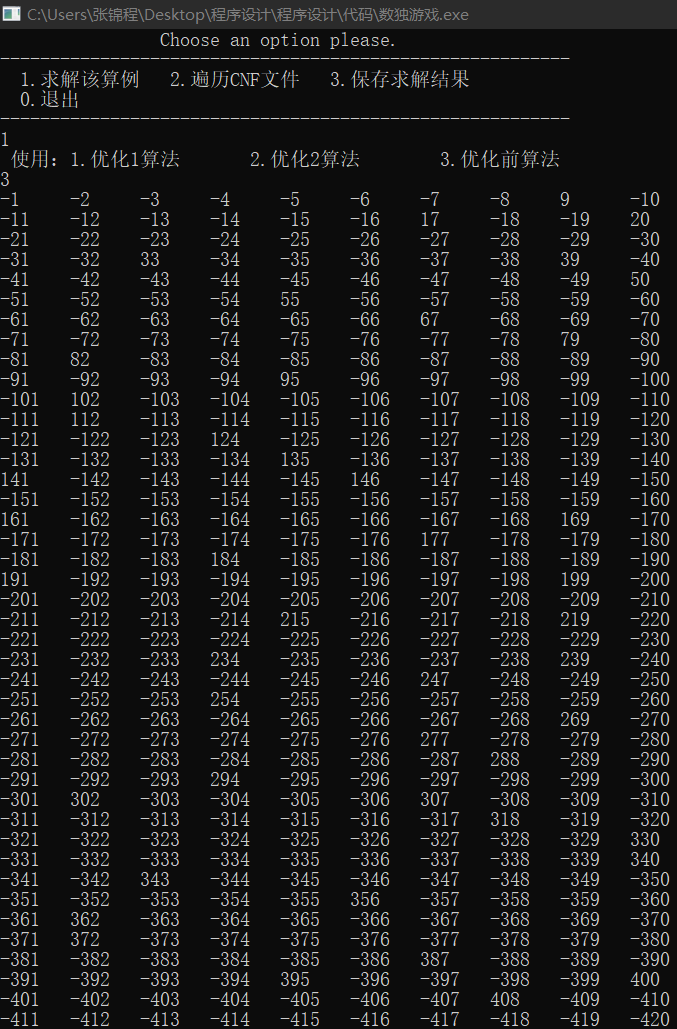


图4.2.2-42 优化前的SAT求解

优化率（1）：[(127-118)/127]\*100%=7.1%

优化率（2）：[(127-350)/127]\*100%=-175.6%

1. 测试算例14：SAT测试备选算例\其它可供选择使用的算例\tst\CBS\_k3\_n100\_m403\_b10\_999.cnf

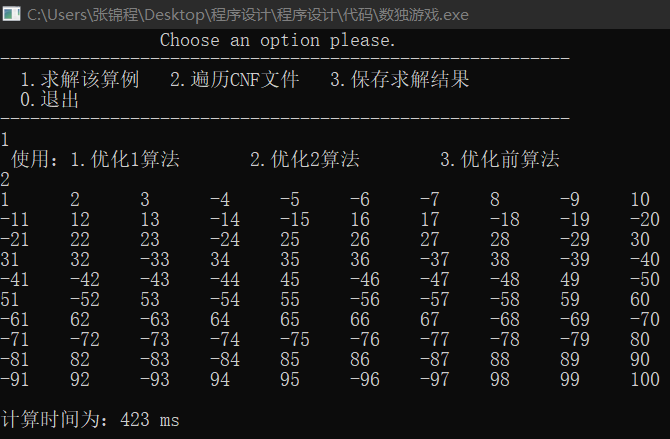
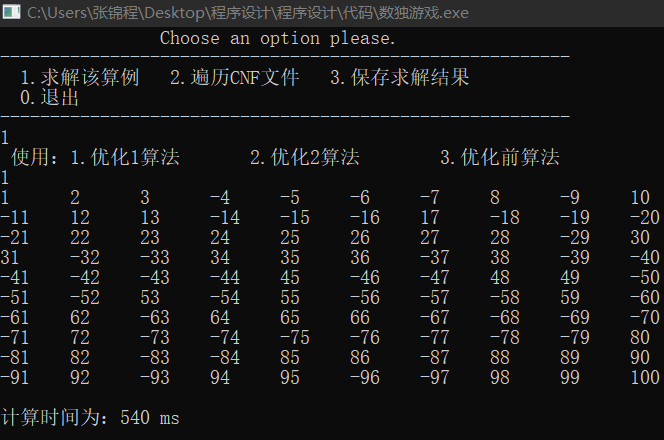


图4.2.2-43 优化1的SAT求解 图4.2.2-44 优化2的SAT求解

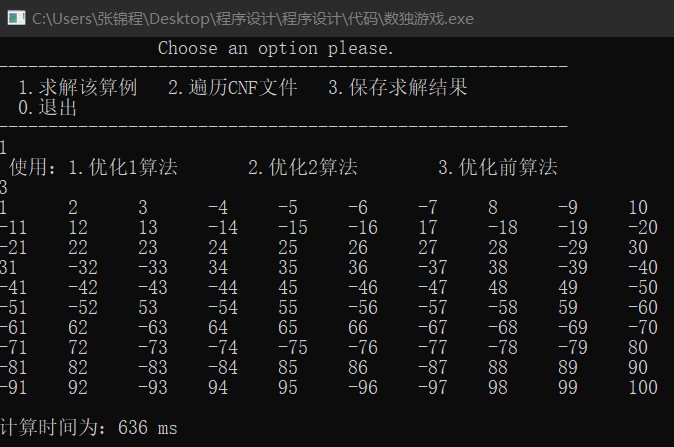


图4.2.2-45 优化前的SAT求解

优化率（1）：[(636-540)/636]\*100%=15.1%

优化率（2）：[(636-423)/636]\*100%=33.5%

1. 测试算例15：SAT测试备选算例\其它可供选择使用的算例\tst\CBS\_k3\_n100\_m403\_b10\_0.cnf，如图所示：

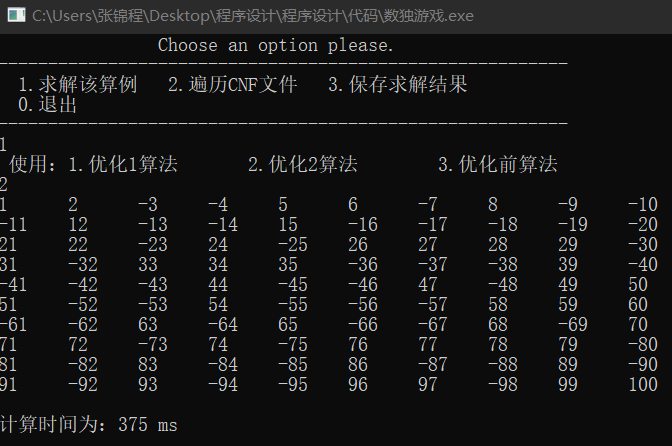
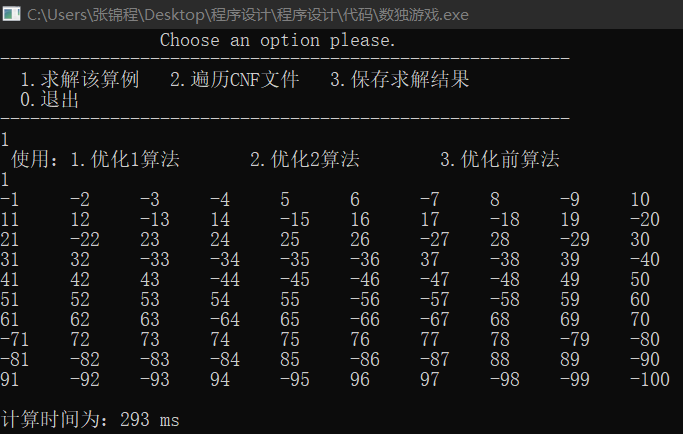


图4.2.2-46 优化1的SAT求解 图4.2.2-47 优化2的SAT求解

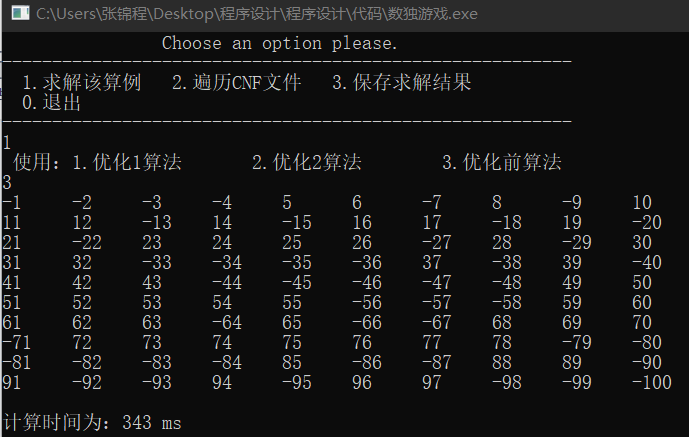


图4.2.2-48 优化前的SAT求解

优化率（1）：[(343-293)/343]\*100%=14.6%

优化率（2）：[(343-375)/343]\*100%=-9.3%

1. 测试算例16：SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v50\_c500.cnf，如图所示：

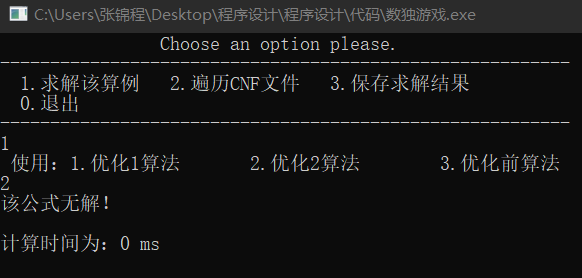
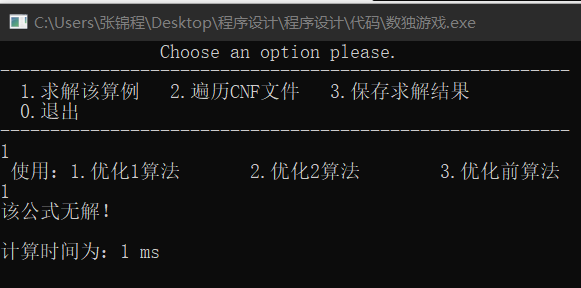


图4.2.2-49 优化1的SAT求解 图4.2.2-50 优化2的SAT求解

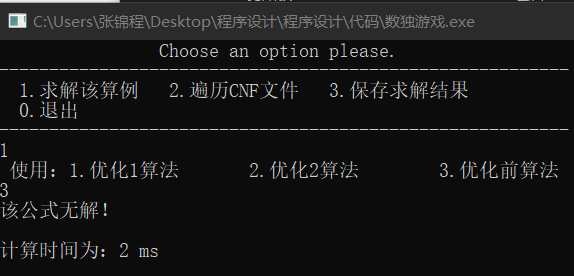


图4.2.2-51 优化前的SAT求解

优化率（1）：[(2-1)/2]\*100%=50%

优化率（2）：[(2-0)/2]\*100%=100%

1. 测试算例17：SAT测试备选算例\其它可供选择使用的算例\ssk.cnf，如图所示：

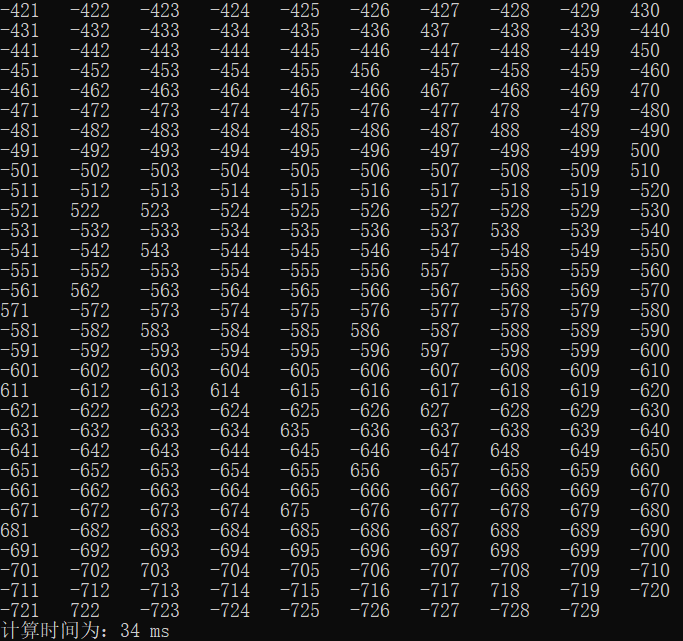
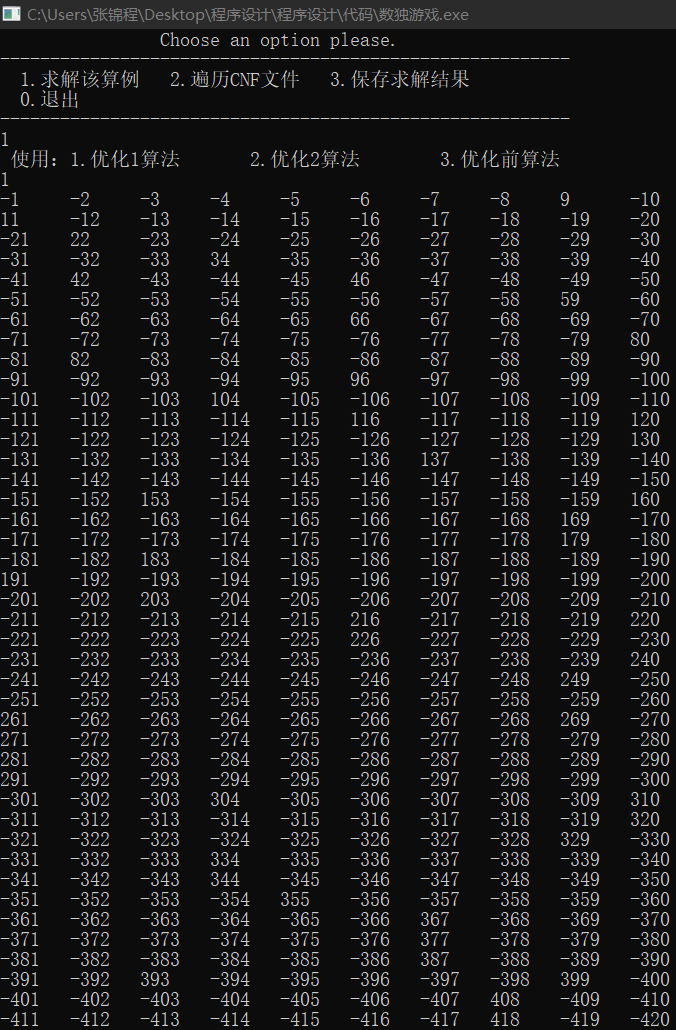


图4.2.2-52 优化1的SAT求解

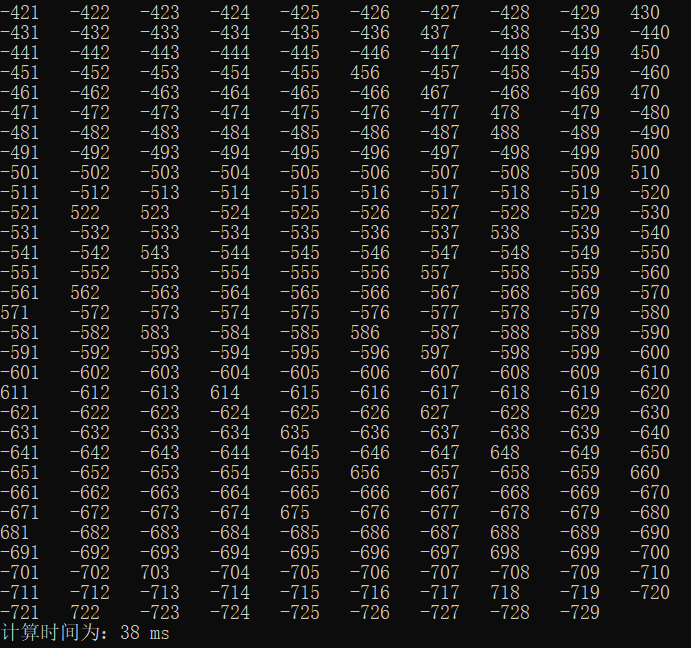
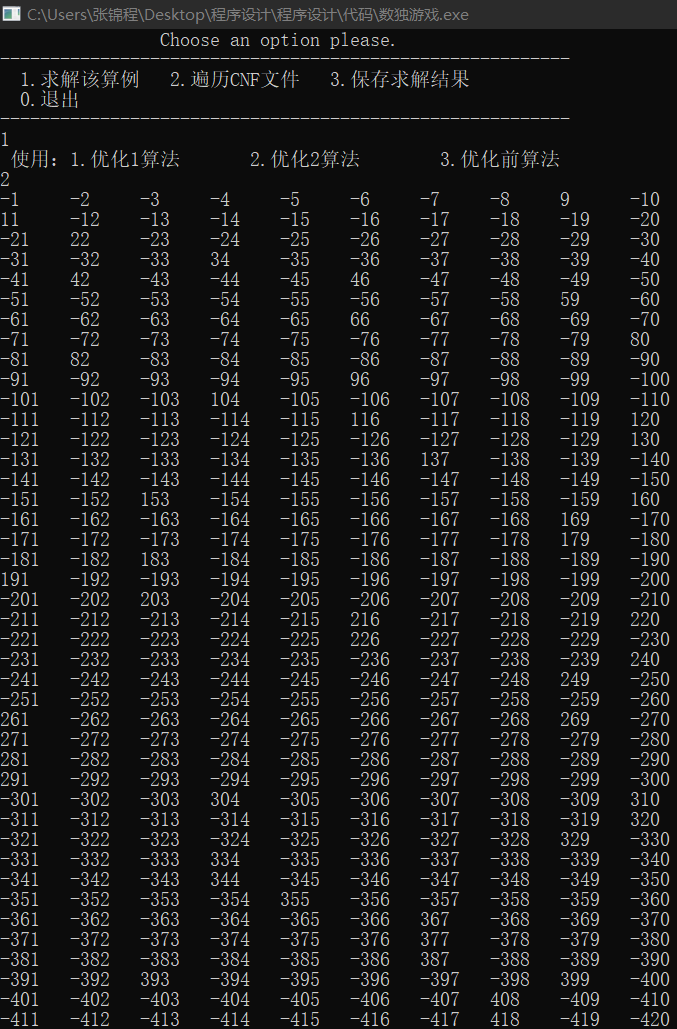


图4.2.2-53 优化2的SAT求解

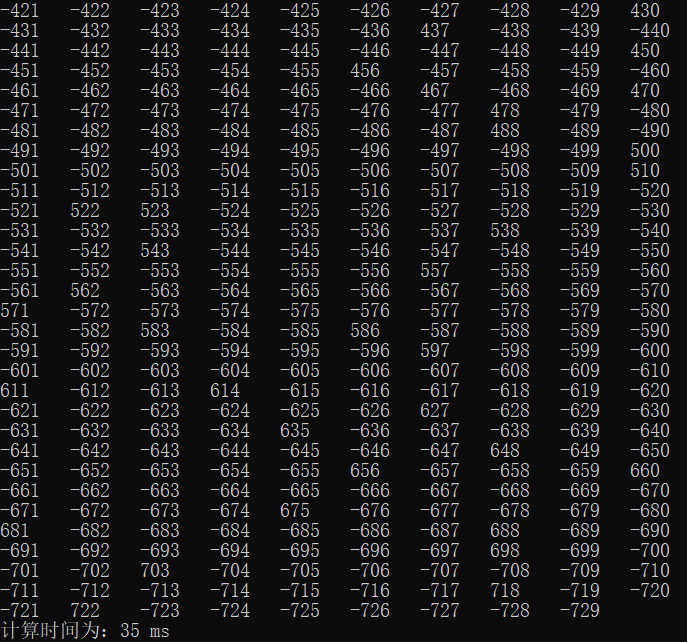
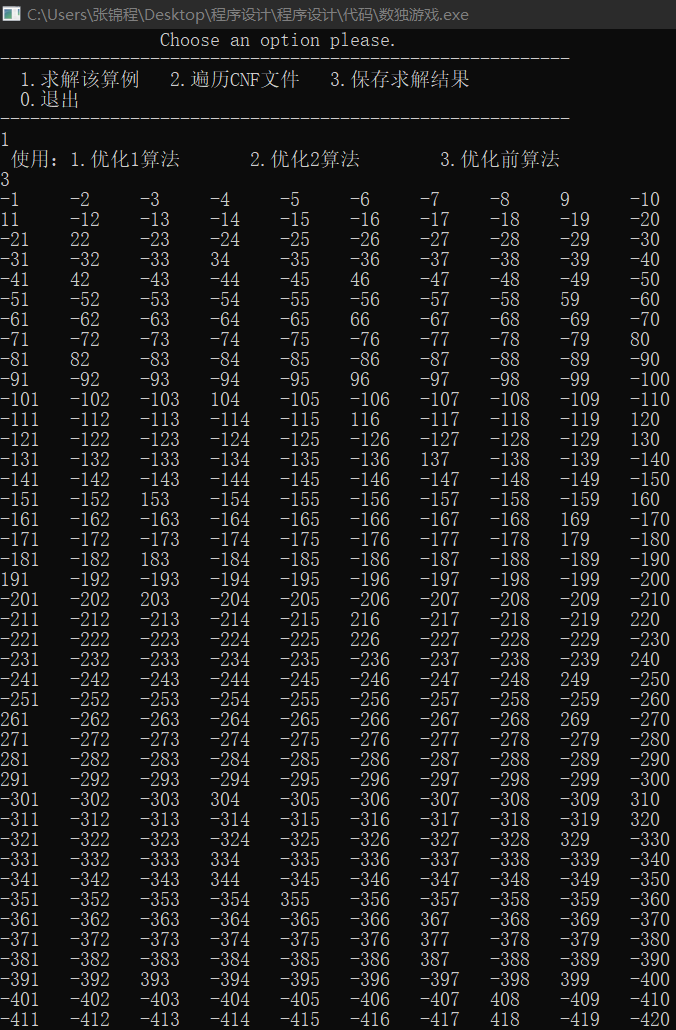


图4.2.2-54 优化前的SAT求解

优化率（1）：[(35-34)/35]\*100%=2.9%

优化率（2）：[(35-38)/35]\*100%=-8.5%

1. 测试算例18：SAT测试备选算例\其它可供选择使用的算例\ss1.cnf

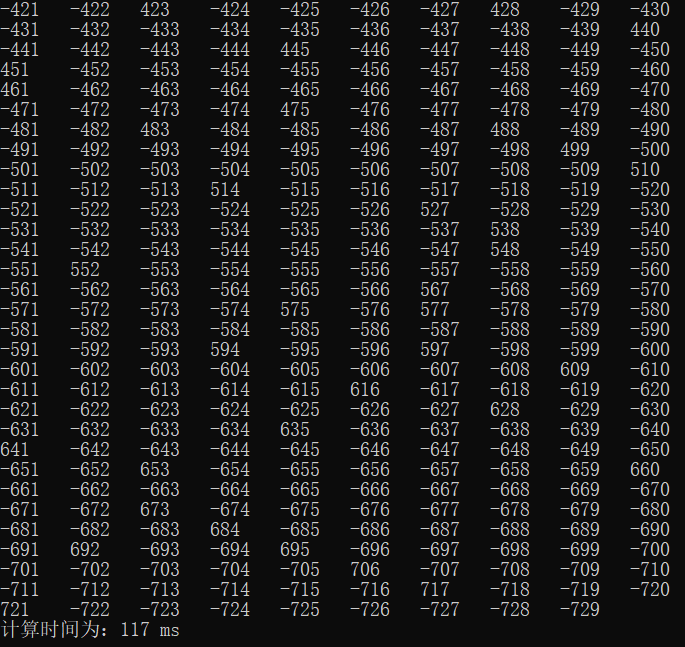
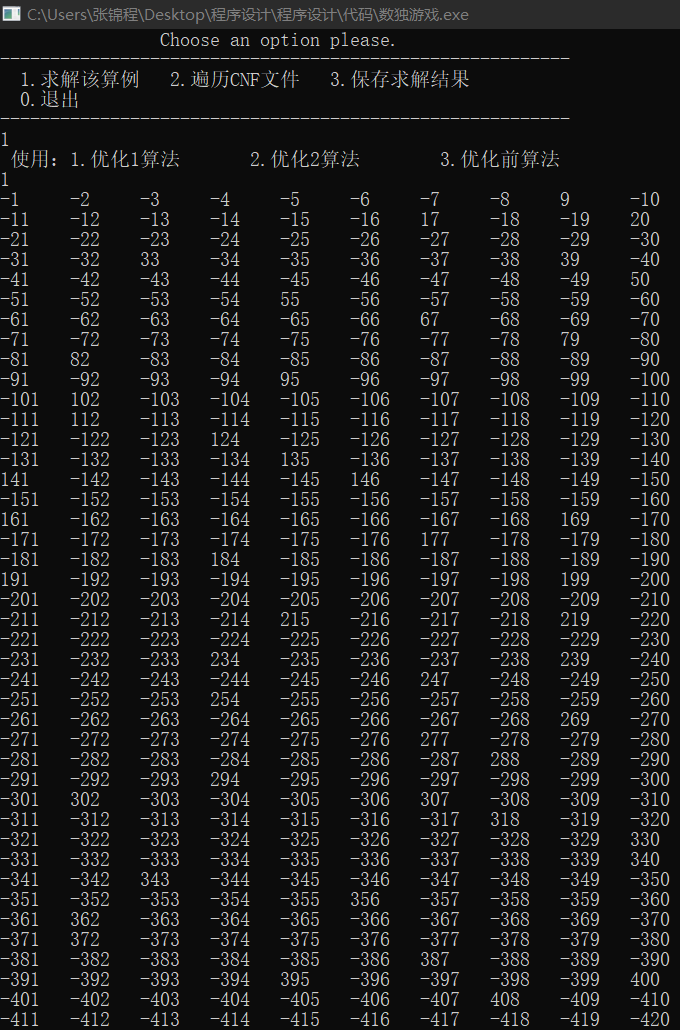


图4.2.2-55 优化1的SAT求解

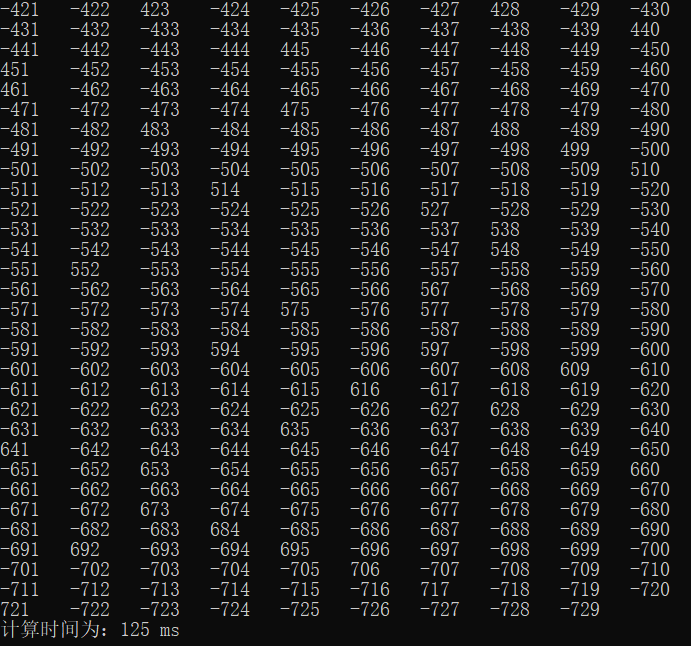
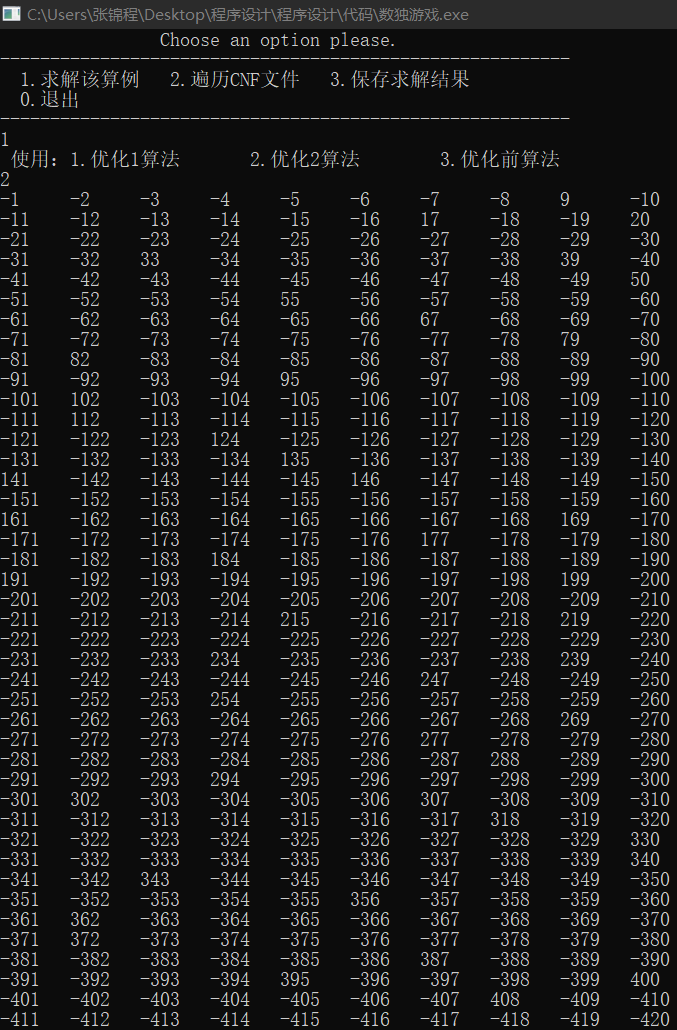


图4.2.2-56 优化2的SAT求解

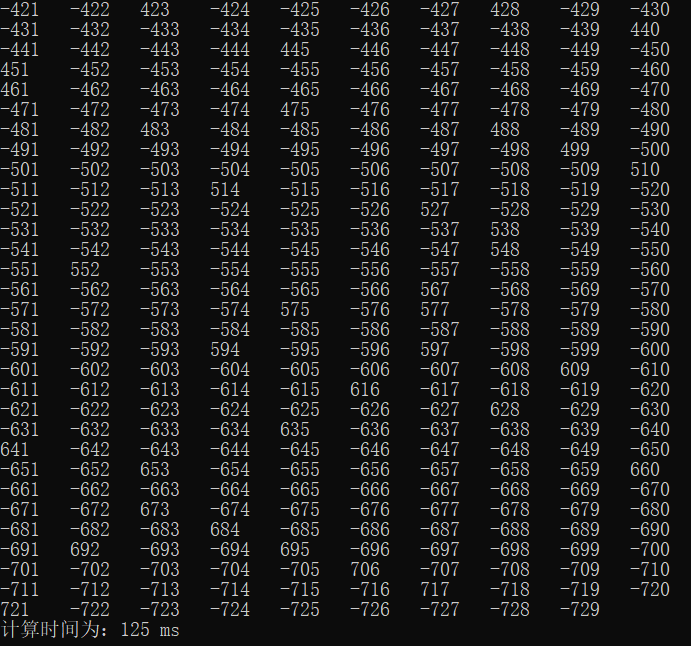
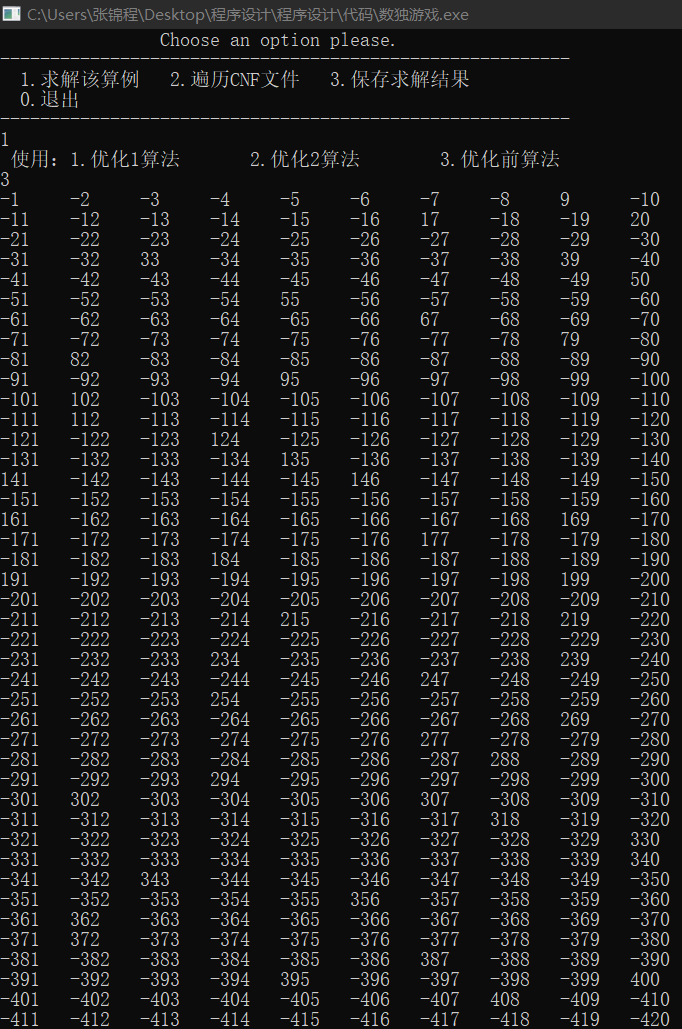


图4.2.2-57 优化前的SAT求解

优化率（1）：[(125-117)/125]\*100%=6.4%

优化率（2）：0

#### 4.2.3 CNF算例测试总结表格

算例测试总结如下表4-1所示：

表4-1 cnf算例测试总结

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 算例名 | 变元数 | 子句数 | 子句数/变元数 | 求解结果 | 求解时间 | 优化率 |
| SAT测试备选算例\其它可供选择使用的算例\problem1.cnf | 20 | 91 | 4.55 | 满足 | 优化前：0ms  优化1：0ms  优化2：0ms | 优化率1：0  优化率2：0 |
| SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v10\_c100.cnf | 10 | 100 | 10 | 不满足 | 优化前：0ms  优化1：0ms  优化2：0ms | 优化率1：0  优化率2：0 |
| SAT测试备选算例\其它可供选择使用的算例\tst\flat30-1.cnf | 90 | 300 | 3.33 | 满足 | 优化前：1ms  优化1：0ms  优化2：0ms | 优化率1：100%  优化率2：100% |
| SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v100\_c400.cnf | 100 | 400 | 4 | 满足 | 优化前：0ms  优化1：0ms  优化2：0ms | 优化率1：0  优化率2：0 |
| SAT测试备选算例\其它可供选择使用的算例\tst\flat30-99.cnf | 90 | 300 | 3.33 | 满足 | 优化前：0ms  优化1：0ms  优化2：0ms | 优化率1：0  优化率2：0 |
| SAT测试备选算例\其它可供选择使用的算例\tst\sw100-1.cnf | 500 | 3100 | 6.2 | 满足 | 优化前：143ms  优化1：8ms  优化2：7ms | 优化率1：94.4%  优化率2：95.1% |
| SAT测试备选算例\其它可供选择使用的算例\ais\ais12.cnf | 265 | 5666 | 21.38 | 满足 | 优化前：114846ms  优化1：  9022ms  优化2：71639ms | 优化率1：92.1%  优化率2：37.6% |
| SAT测试备选算例\满足算例\M\sud00001.cnf | 301 | 2780 | 9.24 | 满足 | 优化前：14ms  优化1：11ms  优化2：11ms | 优化率1：21.4%  优化率2：21.4% |
| SAT测试备选算例\满足算例\M\sud00012.cnf | 232 | 1901 | 8.19 | 满足 | 优化前：11ms  优化1：8ms  优化2：46ms | 优化率1：27.3%  优化率2：-318.2% |
| SAT测试备选算例\满足算例\L\sssc.cnf | 729 | 10319 | 14.16 | 满足 | 优化前：30ms  优化1：29ms  优化2：29ms | 优化率1：3.3%  优化率2：3.3% |
| SAT测试备选算例\其它可供选择使用的算例\tst\qg4-08.cnf | 512 | 9685 | 18.92 | 不满足 | 优化前:2419ms  优化1：2254ms  优化2：2370ms | 优化率1：6.8%  优化率2：2.0% |
| SAT测试备选算例\其它可供选择使用的算例\tst\bw\_large.a.cnf | 459 | 4675 | 10.19 | 满足 | 优化前：852ms  优化1：817ms  优化2：1135ms | 优化率1：4.1%  优化率2：-33.2% |
| SAT测试备选算例\满足算例\M\ssm.cnf | 729 | 10312 | 14.15 | 满足 | 优化前：127ms  优化1：118ms  优化2：350ms | 优化率1：7.1%  优化率2：-175.6% |
| SAT测试备选算例\其它可供选择使用的算例\tst\CBS\_k3\_n100\_m403\_b10\_999.cnf | 100 | 403 | 4.03 | 满足 | 优化前：636ms  优化1：  540ms  优化2：423ms | 优化率1：15.1%  优化率2：33.5% |
| SAT测试备选算例\其它可供选择使用的算例\tst\tst\_v50\_c500.cnf | 50 | 500 | 10 | 不满足 | 优化前：2ms  优化1：1ms  优化2：0ms | 优化率1：50%  优化率2：100% |
| SAT测试备选算例\其它可供选择使用的算例\ssk.cnf | 729 | 10312 | 14.15 | 满足 | 优化前：35ms  优化1：34ms  优化2：38ms | 优化率1：2.9%  优化率2：-8.5% |
| SAT测试备选算例\其它可供选择使用的算例\ss1.cnf | 729 | 10312 | 14.15 | 满足 | 优化前：125ms  优化1：117ms  优化2：125ms | 优化率1：6.4%  优化率2：0。 |
| SAT测试备选算例\其它可供选择使用的算例\tst\CBS\_k3\_n100\_m403\_b10\_0.cnf | 100 | 403 | 4.03 | 满足 | 优化前：343ms  优化1：  293ms  优化2：375ms | 优化率1：14.6%  优化率2：-9.3% |

#### 4.2.4数独游戏求解及简易游玩模块测试

首先简介一下数独部分：分为两个功能，一个是基与挖洞法随机生成数独游戏盘进行交互游玩，并且有选择难度的棋盘生成，游玩过后可以查看答案，第二个是读入数独文件（.txt格式）转化了CNF文件,再DPLL求解CNF文件，最终转化为二维数组打印到屏幕上显示。

1. 数独棋盘的读取（.txt），并且转化为CNF文件进行求解：

.txt的文件内容格式要求是，每行9个字符，未知位置填入“.”，已知位置填入1～9，以回车作为结尾。例如

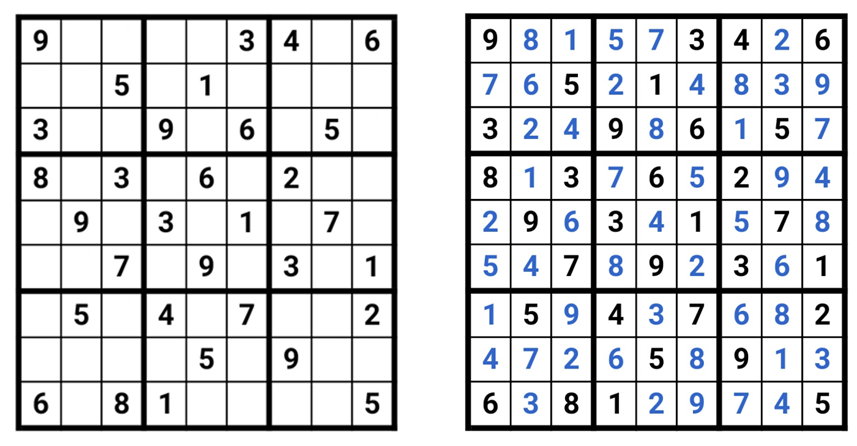


图4.2.4-1 一个Sudoku格局及其解

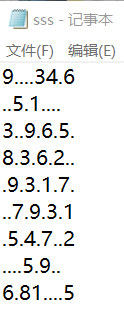


图4.2.4-2 Sudoku输入及其解

1. 随机生成数独棋盘进行游玩（可以选择难度【基于挖洞的数量】），并且可以得到棋盘答案，如图所示：

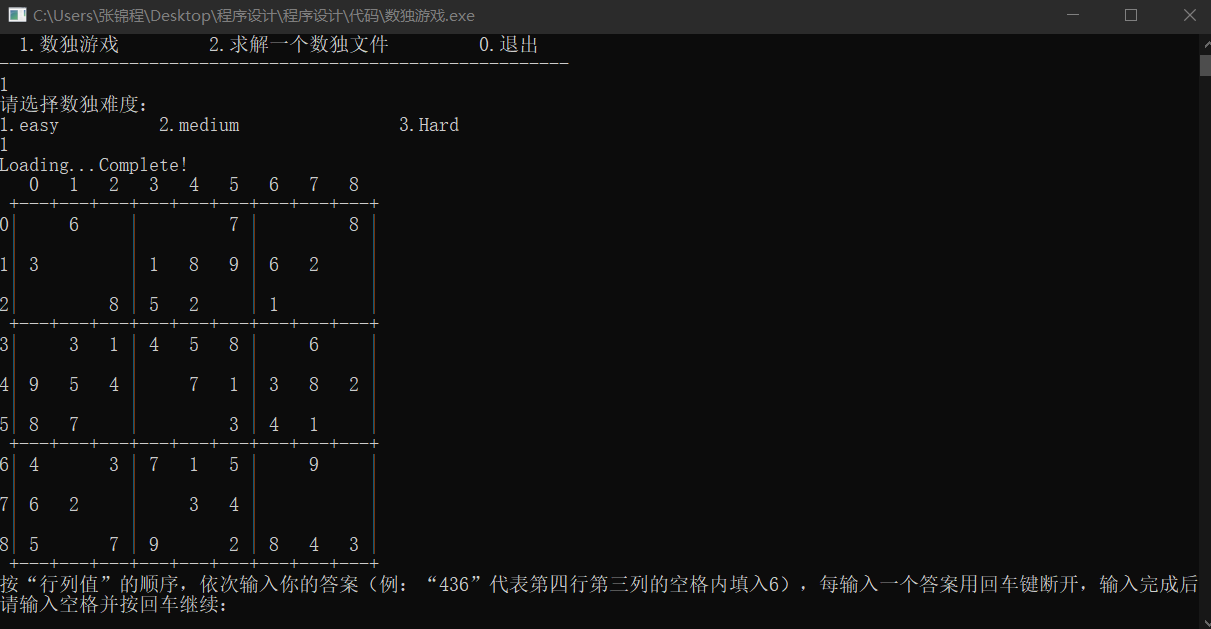


图4.2.4-3 Sudoku随机生成

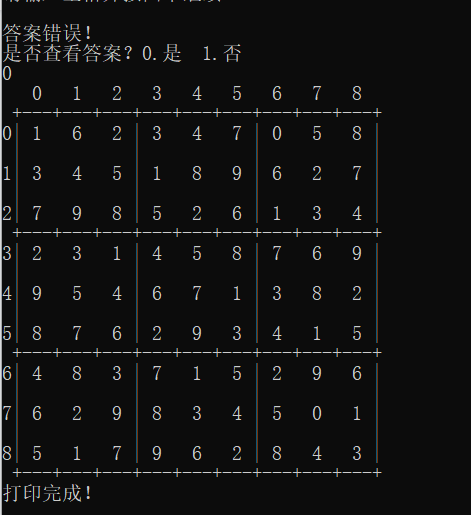


图4.2.4-4 Sudoku随机生成的答案解

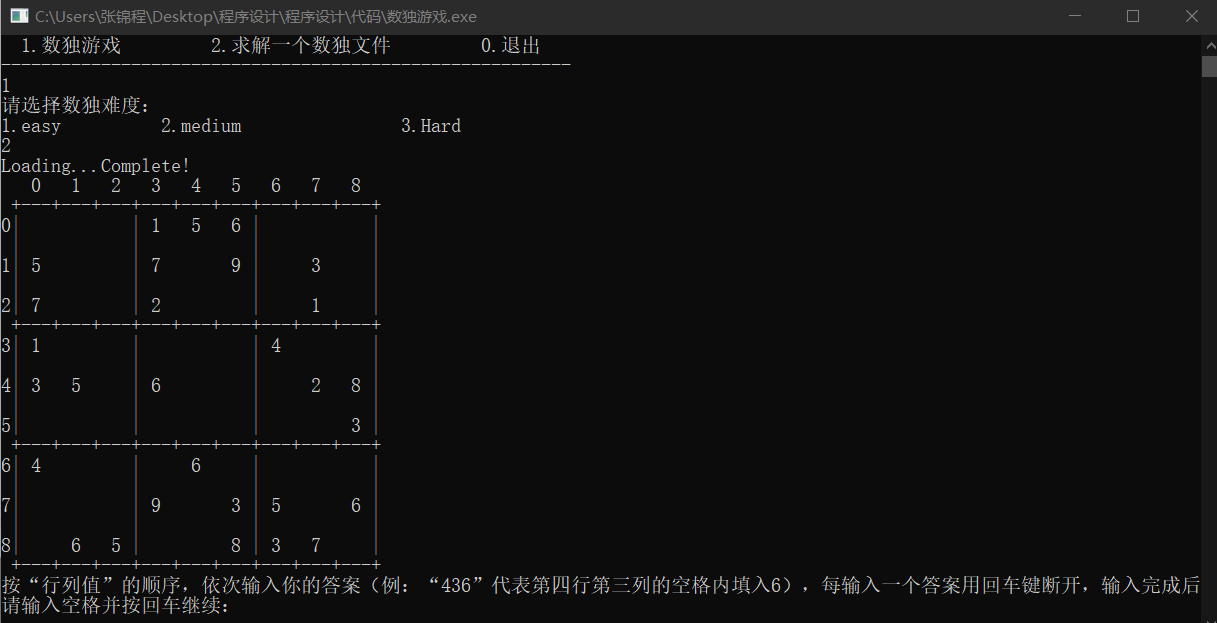


图4.2.4-5 Sudoku随机生成

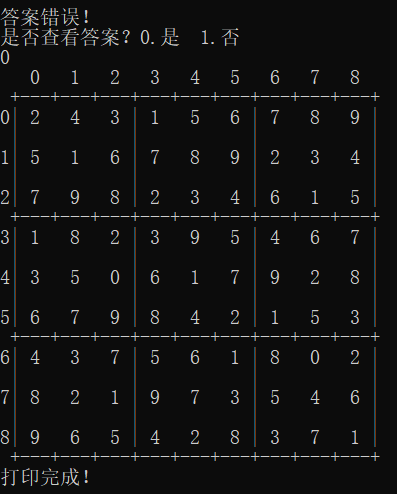


图4.2.4-6 Sudoku随机生成的答案解

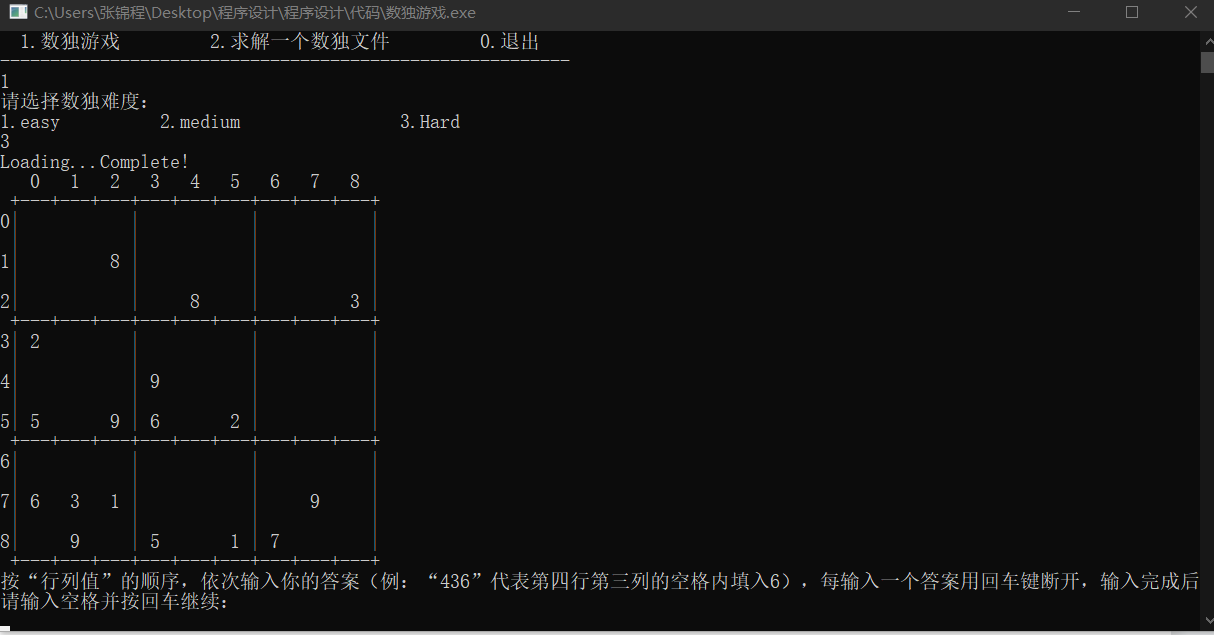


图4.2.4-7 Sudoku随机生成

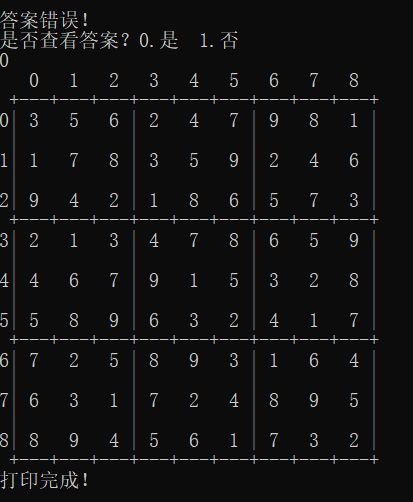


图4.2.4-8 Sudoku随机生成的答案解

**5总结与展望**

**5.1全文总结**

对自己的工作做个总结，主要工作如下：

（1）实现了一个基于DPLL算法的SAT求解器；

（2）在选词策略和判断是否满足的方式上对求解器做了一定优化；

（3）完成了数独游戏的初始化，cnf文件转化，并实现了游戏的可交互性；

（4）设计了测试方案并完成了多个算例的测试；

（5）完善了各种部分的衔接，构建一个简易系统。

**5.2工作展望**

在今后的研究中，围绕着如下几个方面开展工作：

1. 因为设计的SAT求解器较为低级，用了较多的数据结构体去构建。测试过程中，面对较多变元时，可以求解但时间效率往往有点低下。起初仅仅是用链表储存文字和子句，后续优化了才加入变元真值表和正文字负文字信息链表（便于更快的查找到要删除的文字节点），即便如此，时间效率还是有一些低下。在今后的工作中，希望可以得到更多的编程和项目经验；
2. 在DPLL算法实现中，最主要的是变元选择方案。我选择了递归回溯方案，因为设置了较复杂的数据结构体，所以回溯时都要考虑到。变元选择方案也是需要根据变元数和子句数之间的关系。暂未发现如何用一个函数能归纳总结选择如何选择变元方案。在今后的工作中，希望可以得到更多的关于DPLL算法边缘选择编程知识；
3. 在数独游戏模块，我优化了显示界面。我认为我写的程序较为一般，采用了两种补充棋盘方法，一种是横竖遍历求填入不相同的数字，另一种是DPLL求解后根据变元真值表转化为二维数组。挖洞法更是经历了多次失败之后才慢慢摸索出来。在今后的工作中，希望可以通过更多高级语言的学习以及更深入的c语言理解学习到更好的数独游戏交互实现。

# 6体会

在本次课程设计的过程中我收获到了很多编程知识，也认识到了自己的很多不足，比如实现的SAT求解器性能比较低，数独的交互系统设计的也相对简单等等。以下是我按照完成整个课设的时间顺序写的一些体会：

1. 万事开头难。在最开始的时候阅读理解很多次任务书才能较为清晰的感受到大致的方向，比如数独棋盘构建CNF文件的三个约束，将CNF文件中语义编码转换为自然顺序编码的方式。当上手慢慢摸索的时候，经历了多次失败后才慢慢开始懂得如何去构建SAT求解器，DPLL求解和Sudoku系统；
2. 在实现基于DPLL的SAT求解器的过程中，在网上找了很多资料来理解回溯等思想。由于设计了较复杂的结构体，所以回溯的时候也要考虑到，经历了多次失败和屡次调试后才明白问题出错在哪，需要耐心细致的debug。最终实现了DPLL算法，但是变元选择方案的选择还未搞太明白；
3. 在数独游戏中，读了很多资料才知道挖洞法的具体创建，是由已知数独终盘进行挖洞，第一次挖洞肯定是唯一的，所以不用考虑唯一性。之后每次挖洞需要DPLL求解出唯一解，否则挖洞失败。读入txt文件格式转化为二维数组储存，再由二维数组进行创建CNF文件，添加三约束以及每个变元可取1～9的语义序列，再进行DPLL求解。整个过程罗列后慢慢实现，需要很多耐心和细致的观察；
4. 最后就是整合为项目的问题，之前由于没有做课程设计的经验，以为整合为项目很简单，最后在网上看了教程，自己实际操作一遍才发现有很多的问题。主要就是在刚开始写各个模块的时候，数据结构等东西定义的比较随意，导致整合为项目的时候浪费了很多时间，还有一些类似于重复定义的问题，感觉根本无法理解，也是在网上查资料才慢慢理解和解决这些问题。
5. 总而言之，在这次课设实现的过程中收获了非常多的知识，实践出真知的确是真理。代码之路需要很多细致和耐心，路漫漫而其修远兮，吾将上下而求索。

**参考文献**

[1] 张健著. 逻辑公式的可满足性判定—方法、工具及应用. 科学出版社，2000

[2]Tanbir Ahmed. An Implementation of the DPLL Algorithm. Master thesis, Concordia University,Canada,2009

[3] 陈稳. 基于DPLL的SAT算法的研究与应用.硕士学位论文，电子科技大学，2011

[4]Carsten Sinz. Visualizing SAT Instances and Runs of the DPLL Algorithm. J Autom Reasoning (2007) 39:219–243

[5]360百科：数独游戏<https://baike.so.com/doc/3390505-3569059.html>

[6] Tjark Weber. A sat-based sudoku solver. In 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15, 2005.

[7] Ins Lynce and Jol Ouaknine. Sudoku as a sat problem. In Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale. Springer, 2006.

[8] Uwe Pfeiffer, Tomas Karnagel and Guido Scheffler. A Sudoku-Solver for Large Puzzles using SAT. LPAR-17-short (EPiC Series, vol. 13), 52–57

[9] Sudoku Puzzles Generating: from Easy to Evil.

http://zhangroup.aporc.org/images/files/Paper\_3485.pdf

[10] Robert Ganian and Stefan Szeider. Community Structure Inspired Algorithms for SAT and #SAT. SAT 2015, 223-237360

**附录**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define INCREASEMENT 100

typedef int status;

/\*定义子句链表结点结构类型\*/

typedef struct Clause{

int literal;//记录子句中的文字

int flag;

//标记该文字是否已被删除，未删除时值为0，否则值为使之删除的变元序号便于回复

struct Clause \*nextl;//指向该子句中下一文字的指针

struct Clause \*litline;

//指向整个CNF公式邻接链表中下一个文字相同的子句结点，用于回复与方便查找

}Clause;

/\*定义CNF范式链表结点（即子句链表头结点）结构类型\*/

typedef struct Paradigm{

int number;//子句中显示的文字数，便于找单子句

int flag;

//标记该子句是否已被删除，未删除时值为0，否则值为使之删除的变元序号便于回复

struct Paradigm \*nextc;//指向下一子句的头结点

struct Clause \*sentence;//子句头指针

}Paradigm;

/\*定义CNF范式链表头结点类型，存储CNF范式信息\*/

typedef struct Root{

int litsize;//存储文字数量

int parasize;//存储子句数量

Paradigm \*first;//指向第一个子句

}Root;

//Dpll会用到的两个结构，用于更快寻找正文字和负文字

/\*定义指向子句链表头结点的指针链表结点结构类型\*/

typedef struct Paraline{

Paradigm \*claline;//指向子句链表头结点Paradigm

struct Paraline \*next;//指向下一链表结点

} Paraline;

/\*定义文字相关信息链表结构类型\*/

typedef struct LitTrabverse{

Paraline \*Tra\_cla;//指向含有该正文字或负文字的子句头结点链表的头结点

Clause \*Tra\_lit;//指向该正文字或负文字的文字结点

}LitTrabverse;

/\*定义求解数独文件时所需的存储变元信息结构类型\*/

typedef struct sudokusolver{

int x;//存储行信息

int y;//存储列信息

int z;//存储变元对应1～9数值信息

} sudokusolver;

/\*定义存储变元信息的变元线性表结点结构类型\*/

typedef struct ArgueValue{

int Value;//变元的真值

int IsInit;//变元是否已赋值

int Occur\_times;//变元在所有子句中出现的总次数，用于分裂政策的选择

LitTrabverse Pos;//变元所有正文字的相关信息结构

LitTrabverse Neg;//变元所有负文字的相关信息结构

sudokusolver xyz;//求解数独文件时所需的变元信息

} ArgueValue;

//全局变量

char filename[1000]; //文件名

long instacksize;//记录栈stack的大小

ArgueValue \*ValueList;//变元真值表

Root \*r;//链表

int \*Frequency;//变元出现次数辅助数组

int \*stack;//DPLL辅助栈

int sudoku\_table[9][9];//定义全局变量int类型二维数组存储数独盘

int users\_sudoku[9][9];//存储输出的含空格数独问题格局

int sum\_op;//总选择

/\*------------------------CNF文件处理------------------------\*/

status CreateParadigm(FILE \*\*fp);//创建CNF范式邻接链表及变元表

int CreateClause(FILE \*\*fp,Clause \*\*sentence,Paradigm \*ClausHead,int first);//创建子句链表及文字链表

status DestroyParadigm(Root \*r);//销毁所有链表及线性表结构

int HasUnitClause(Root \*r);//判断CNF范式中是否还含有单子句

Clause \* HasUnitClause\_Before(Root \*r);//判断CNF范式中是否还含有单子句（优化前版本）

status isUnitClause(Paradigm \*c);//判断指针c指向的子句链表是否为单子句链表

/\*在整个CNF公式中取一个文字\*/

int FindLiteral1(Root \*r);//取每次DPLL处理后公式中Occur\_Times最大的文字

int FindLiteral2(Root \*r);//取原公式中Occur\_Times最大的文字

int FindLiteral3(Root \*r);//取子句中第一个未被删除（flag为0）的文字

int FindLiteral4(Root \*r);//变元表正序正序第一个未赋真值（IsInit=0）的变元的正文字

Clause \* FindLiteral\_Before(Root \*r);//在整个CNF公式中取一个文字（优化前版本）

status DeleteClause(Root \*r,int l);//删除出现了文字l的所有单子句

status AddClause(Root \*r,int l);//在CNF范式邻接链表表头添加只含有文字l的单子句链表

status RemoveHeadClaus(Root \*r,int l);//删除CNF范式邻接链表中从表头开始第一个只含有文字l的单子句链表

status DeleteLiteral(Root \*r,int l);//删除所有文字为-l的子句链表结点

status RecoverCNF(Root \*r,int l);//恢复认为文字l为真时对CNF范式邻接链表所作的操作

void ParadigmTrabverse(Root \*r);//遍历CNF范式邻接链表

status SaveValue(ArgueValue \*ValueList,int solut,int time);//保存CNF范式的解及求解时间信息

/\*------------------------DPLL算法------------------------\*/

status SAT(void);

status DPLL1(int num,int op,int times);//采用第一种变元选取策略的递归算法DPLL函数

status DPLL2(int num,int op,int times);//采用第二种变元选取策略的递归算法DPLL函数

status DPLL3(int num,int op,int times);//采用第三种变元选取策略的递归算法DPLL函数

status DPLL4(int num,int op,int times);//采用第四种变元选取策略的递归算法DPLL函数

status DPLL\_Before(Root \*r,int op);//优化前版本DPLL

void AnswerComplete(void);//完善SAT求解结果

status AnswerCheck(int solut);//检查SAT求解结果正确性

/\*------------------------SuDoKu------------------------\*/

status Sudoku(void);

FILE \* CreateSudokuFile(void);//创建数独问题转化为SAT问题后的cnf文件

status CreateSudoku(void);//创建新的随机9\*9数独终盘

status SolveSudoku(void);//求解数独cnf文件并转化为数独终盘形式输出

status DigHole(int num);//对已知数独终盘挖num个洞生成有唯一解的数独问题

status SudokuComplete(void);//完善数独终盘

FILE \* SetCNFfile(void);//将数独txt文件转化为cnf

status CNFSudokuTableTransform(void);//数独对应SAT变元表转化为二维数组

status SudokuTablePrint(void);//输出数独盘

/\*------------------------CNF文件处理------------------------\*/

/\*创建CNF范式邻接链表及变元表

参数：(FILE\*\*)open：文件的指针地址\*/

status CreateParadigm(FILE \*\*fp) {

char readfile[20];//定义字符类型数组记录在文件中读到的内容

int l,i,j;

Paradigm \*p,\*q;

r=(Root\*)malloc(sizeof(Root));

r->first=NULL;

while((fscanf(\*fp,"%s",readfile)!=EOF)!=EOF) { //循环读文件

if (strcmp(readfile,"p")==0) //读到‘p’说明开始读文件

break;

}

while (fscanf(\*fp, "%s",readfile)!=EOF) {

if (strcmp(readfile,"cnf")==0) {//从文件中读到字符串“cnf”

fscanf(\*fp, "%d",&l);

r->litsize=l;//读取CNF文件 变元数 存入r->litsize

fscanf(\*fp, "%d",&l);

r->parasize=l;//读取CNF文件 子句总数 存入r->parasize

break;

}

}

if(sum\_op==1) printf("文字数%d ，子句数%d！\n",r->litsize,r->parasize);

//创建变元表（初始化）

ValueList=(ArgueValue\*)malloc((r->litsize+1)\*sizeof(ArgueValue));

if (ValueList==NULL) return OVERFLOW;//没分配成功

for (i=0; i<=r->litsize; i++) {

ValueList[i].IsInit=0;

ValueList[i].Occur\_times=0;

ValueList[i].Pos.Tra\_cla=NULL;

ValueList[i].Pos.Tra\_lit=NULL;

ValueList[i].Neg.Tra\_cla=NULL;

ValueList[i].Neg.Tra\_lit=NULL;

}

Frequency=(int \*)malloc((r->litsize)\*sizeof(int));

i=2;

if (r->parasize!=0) {

//创建第一个子句头结点链表结点

if (fscanf(\*fp, "%d",&l)!=EOF&&l!=0) {

p=(Paradigm\*)malloc(sizeof(Paradigm));

if (p==NULL) return OVERFLOW;

r->first=p;//头节点

q=p;

p->number=CreateClause(fp, &p->sentence, p, l);//创建其对应子句链表

p->flag=0;

p->nextc=NULL;

//创建CNF范式链表

while (i<=r->parasize) {

i++;

fscanf(\*fp, "%d",&l);

p=(Paradigm\*)malloc(sizeof(Paradigm));

if (p==NULL) return OVERFLOW;

p->number=CreateClause(fp, &p->sentence, p, l);

p->flag=0;

p->nextc=NULL;

q->nextc=p;

q=q->nextc;

}

}

}

//创建栈stack记录DPLL过程中设为1的文字及随之产生的单子句文字(后续用DPLL)

stack=(int\*)malloc(20000\*sizeof(int));

stack[0]=0;

instacksize=0;

/\*将变元按出现次数由多至少在Frequency数组中顺序排列\*/

for (i=0; i<r->litsize; i++)

Frequency[i]=i+1;

for (i=0; i<r->litsize; i++) {

for (j=i+1; j<=r->litsize; j++) {

if (ValueList[Frequency[i]].Occur\_times<ValueList[Frequency[j-1]].Occur\_times) {

l=Frequency[i];

Frequency[i]=Frequency[j-1];

Frequency[j-1]=l;

}

}

}

fclose(\*fp);

return OK;

}

/\*创建子句链表及文字链表

参数：(FILE\*\*)open：文件的指针地址

(Clause \*\*)sentence:子句的头结点 Clause类型指针地址

(Paradigm \*)ClausHead：子句链表头节点指针值

(int )first：创建的子句的第一个文字的值

返回值：i：子句内文字的数量

ERROR：该子句为空子句

OVERFLOW：空间不足溢出\*/

int CreateClause(FILE \*\*fp,Clause \*\*sentence,Paradigm \*ClausHead,int first) {

Clause \*p,\*q;

Paraline \*pline;

int l,l1,i=0;//i记录单个子句中文字数量

if (first==0) return ERROR;//不存在没有文字的子句

p=(Clause\*)malloc(sizeof(Clause));

if (p==NULL) return OVERFLOW;

p->literal=first;

p->flag=0;

p->nextl=NULL;

p->litline=NULL;

q=p;

\*sentence=p;

i++;

l1=abs(first);

ValueList[l1].Occur\_times++;

if (first>0) { //用来DPLL时寻找相同文字的不同子句以及删除后回复

p->litline=ValueList[l1].Pos.Tra\_lit;

ValueList[l1].Pos.Tra\_lit=p;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=ClausHead;

pline->next=ValueList[l1].Pos.Tra\_cla;

ValueList[l1].Pos.Tra\_cla=pline;

} else { //同上

p->litline=ValueList[l1].Neg.Tra\_lit;

ValueList[l1].Neg.Tra\_lit=p;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=ClausHead;

pline->next=ValueList[l1].Neg.Tra\_cla;

ValueList[l1].Neg.Tra\_cla=pline;

}

fscanf(\*fp, "%d",&l);

while (l!=0) {

// 创建文字链表结点

p=(Clause\*)malloc(sizeof(Clause));

if (p==NULL) return OVERFLOW;

p->literal=l;

p->flag=0;

p->nextl=NULL;

q->nextl=p;

q=q->nextl;

i++;

// 创建变元链表

l1=abs(l);

ValueList[l1].Occur\_times++;

if (l>0) {

p->litline=ValueList[l1].Pos.Tra\_lit;

ValueList[l1].Pos.Tra\_lit=p;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=ClausHead;

pline->next=ValueList[l1].Pos.Tra\_cla;

ValueList[l1].Pos.Tra\_cla=pline;

} else {

p->litline=ValueList[l1].Neg.Tra\_lit;

ValueList[l1].Neg.Tra\_lit=p;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=ClausHead;

pline->next=ValueList[l1].Neg.Tra\_cla;

ValueList[l1].Neg.Tra\_cla=pline;

}

fscanf(\*fp, "%d",&l);//读取下一文字

}

return i;

}

/\*销毁CNF范式邻接链表\*/

status DestroyParadigm(Root \*r) {

Paradigm \*p;

Clause \*t;

Paraline \*pline;

int i;

if (r->first!=NULL) {

p=r->first;

//销毁CNF范式链表

while (p!=NULL) {

if (p->sentence!=NULL) {

t=p->sentence;

//销毁单个子句链表

while (t!=NULL) {

p->sentence=t->nextl;

free(t);//释放存储文字的单个子句链表结点空间

t=p->sentence;

}

}

r->first=p->nextc;

free(p);//释放子句链表头结点存储空间

p=r->first;

}

}

//释放变元正负文字信息链表存储空间

for (i=0; i<=r->litsize; i++) {

for (pline=ValueList[i].Pos.Tra\_cla; pline!=NULL; pline=ValueList[i].Pos.Tra\_cla) {

ValueList[i].Pos.Tra\_cla=pline->next;

free(pline);

}

ValueList[i].Pos.Tra\_lit=NULL;

for (pline=ValueList[i].Neg.Tra\_cla; pline!=NULL; pline=ValueList[i].Neg.Tra\_cla) {

ValueList[i].Neg.Tra\_cla=pline->next;

free(pline);

}

ValueList[i].Neg.Tra\_lit=NULL;

}

free(Frequency);

free(r);

return OK;

}

/\*判断CNF范式中是否存在单子句

返回类型：int

返回值：存在单子句：该单子句所含的唯一文字

不存在单子句：0\*/

int HasUnitClause(Root \*r) {

Paradigm \*p;

Clause \*t;

for(p=r->first; p!=NULL; p=p->nextc) {

if(p->flag==0) { //子句没有被删除

if(p->number==1)//子句文字数为1;

for (t=p->sentence; t!=NULL; t=t->nextl) {

if (t->flag==0)//文字没有被删除

return t->literal;

}

}

}

return 0;

}

/\*判断CNF范式中是否还含有单子句（优化前版本）

返回类型：Clause \*

返回值：存在单子句：该单子句所含的唯一Clause结点的指针

不存在单子句：NULL\*/

Clause \* HasUnitClause\_Before(Root \*r) {

Paradigm \*p;

Clause \*t;

for(p=r->first; p!=NULL; p=p->nextc) {

if(p->flag==0) { //子句没有被删除

if(p->number==1)//子句文字数为1;

for (t=p->sentence; t!=NULL; t=t->nextl) {

if (t->flag==0)//文字没有被删除

return t;

}

}

}

}

/\*判断参数指针c指向的子句头结点对应子句是否为单子句；

返回值：TRUE：是单子句；

ERROR：不是单子句\*/

status isUnitClause(Paradigm \*c) {

if (c->number==1)

return TRUE;

else return FALSE;

}

/\*在整个CNF公式中取一个文字\*/

/\*way1: 取每次DPLL处理后公式中出现次数最多（Occur\_Times值最大）的文字

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0\*/

int FindLiteral1(Root \*r) {

int i,num=0,flag=0;

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].IsInit==0) {

num=i;//找到变元表正序第一个未被赋真值的变元，num记录其序号

flag=1;//表示变元表内存在未被赋真值的变元

break;

}

}

if (flag==1) {

for (i++; i<=r->litsize; i++) {

if (ValueList[i].IsInit==0) {

if (ValueList[i].Occur\_times>ValueList[num].Occur\_times)

num=i;

}

}

}

return num;

}

/\*way2:取原公式中未赋真值的变元中出现次数最多（Occur\_Times最大）的变元正文字

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0\*/

int FindLiteral2(Root \*r) {

int i=0;

for (i=0; i<r->litsize; i++) {

if (ValueList[Frequency[i]].IsInit==0) {

return Frequency[i];

}

}

return 0;

}

/\*way3:取子句中第一个未被删除（flag为0）的文字

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0\*/

int FindLiteral3(Root \*r){

Paradigm \*p;

Clause \*c;

for (p=r->first; p!=NULL; p=p->nextc)

if (p->flag==0) {//子句未被删除

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {//文字未被删除

return c->literal;

}

}

}

return 0;

}

/\*way4: 取变元表正序正序第一个未赋真值（IsInit=0）的变元的正文字

返回类型：int

返回值：公式非空：选中文字的值

公式中没有剩余文字：0\*/

int FindLiteral4(Root \*r){

int i=0;

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].IsInit==0) {

return i;

}

}

return 0;

}

/\*在整个CNF公式中取一个文字（优化前版本）：取子句中第一个未被删除（flag为0）的文字

返回类型：Clause \*

返回值：公式非空：选中文字的结点指针

公式中没有剩余文字：NULL\*/

Clause \* FindLiteral\_Before(Root \*r) {

Paradigm \*p;

Clause \*c;

for (p=r->first; p!=NULL; p=p->nextc) {

if (p->flag==0) {

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {

return c;

}

}

}

}

return NULL;

}

/\*删除含参数l的子句

参数l：真值为1的文字（int）

(此函数不应使公式中的子句数量减少）\*/

status DeleteClause(Root \*r,int l) {

int l1;

Paradigm \*p;

Clause \*c;

Paraline \*pline;

l1=abs(l);

if(l>0) {

//l为正文字情况，搜索l对应变元的正文字信息链表

for(pline=ValueList[l1].Pos.Tra\_cla; pline; pline=pline->next) {

p=pline->claline;

if(p->flag==0) { //p指向的子句头结点对应子句未被删除

p->flag=l1;//将p指向结点的flag值标记为l，表示该子句因文字l真值为1而被删除

r->parasize--;

for (c=p->sentence; c!=NULL; c=c->nextl)//依次修改p指向子句的flag标记值

if (c->flag==0) {

c->flag=l1;

ValueList[abs(c->literal)].Occur\_times--;

p->number--;

}

}

}

} else {

//l为负文字情况，搜索l对应变元的负文字信息链表

for (pline=ValueList[l1].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->flag==0) {

p->flag=l1;

r->parasize--;

for (c=p->sentence; c!=NULL; c=c->nextl)

if (c->flag==0) {

c->flag=l1;

ValueList[abs(c->literal)].Occur\_times--;

p->number--;

}

}

}

}

return OK;

}

/\*删除所有文字为-l的子句链表结点

参数l：(int)真值为1的文字；

返回值：OK：成功删除文字；

FALSE：公式无解

(此函数不应使公式中的子句数量减少）\*/

status DeleteLiteral(Root \*r,int l) {

Paradigm \*p;

Clause \*c;

Paraline \*pline;

int l1,l\_op;

l1=abs(l);

l\_op=0-l;

if (l>0) {

for (pline=ValueList[l1].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->flag==0) {//p指向的子句链表头结点对应的子句未被删除

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->literal==l\_op) {

if (c->flag==0) {//c指向子句链表结点内文字未被删除

if (p->number==1) return FALSE;//要删除的文字为当前子句内的唯一文字，该子句真值为0，公式无解

c->flag=l1;//标记该结点内flag值为l1，表示因l真值为1而被删除

ValueList[abs(c->literal)].Occur\_times--;//变元表内该变元出现的次数减一

p->number--;

}

}

}

}

}

} else {

for (pline=ValueList[l1].Pos.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->flag==0) {

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->literal==l\_op) {

if (c->flag==0) {

if (p->number==1) return FALSE;

c->flag=l1;

ValueList[abs(c->literal)].Occur\_times--;

p->number--;

}

}

}

}

}

}

return OK;

}

/\*在CNF范式邻接链表表头添加只含有文字l的单子句链表

参数l：(int)增加的单子句内的唯一文字

返回值：OVERFLOW：空间不足溢出；

OK：子句增加成功\*/

status AddClause(Root \*r,int l) {

Paradigm \*p;

Clause \*c;

Paraline \*pline;

int l1;

p=(Paradigm\*)malloc(sizeof(Paradigm));

if (p==NULL) return OVERFLOW;

p->nextc=r->first;

c=(Clause\*)malloc(sizeof(Clause));

if (c==NULL) return OVERFLOW;

c->literal=l;

c->flag=0;

c->nextl=NULL;

l1=abs(l);

if (l>0) {

//l为正文字，在变元表内l对应变元的正文字信息链表表头增加新增结点信息

c->litline=ValueList[l1].Pos.Tra\_lit;

ValueList[l1].Pos.Tra\_lit=c;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=p;

pline->next=ValueList[l1].Pos.Tra\_cla;

ValueList[l1].Pos.Tra\_cla=pline;

} else {

//l为负文字，在变元表内l对应变元的负文字信息链表表头增加新增结点信息

c->litline=ValueList[l1].Neg.Tra\_lit;

ValueList[l1].Neg.Tra\_lit=c;

pline=(Paraline\*)malloc(sizeof(Paraline));

pline->claline=p;

pline->next=ValueList[l1].Neg.Tra\_cla;

ValueList[l1].Neg.Tra\_cla=pline;

}

p->sentence=c;

p->number=1;

p->flag=0;

r->first=p;

r->parasize++;

return OK;

}

/\*删除CNF范式邻接链表中从表头开始第一个只含有文字l的单子句链表

参数l：(int)要删除的单子句的唯一文字值；

返回值：OK：删除成功\*/

status RemoveHeadClaus(Root \*r,int l) {

Paradigm \*p,\*q;

Clause \*c;

Paraline \*pline;

int l1;

l1=abs(l);

if (l>0) {

//l为正文字，要删除的结点信息必在变元表内l对应变元的正文字信息链表表头

pline=ValueList[l1].Pos.Tra\_cla;

p=pline->claline;

ValueList[l1].Pos.Tra\_cla=pline->next;

ValueList[l1].Pos.Tra\_lit=ValueList[l1].Pos.Tra\_lit->litline;

} else {

//l为负文字，要删除的结点信息必在变元表内l对应变元的负文字信息链表表头

pline=ValueList[l1].Neg.Tra\_cla;

p=pline->claline;

ValueList[l1].Neg.Tra\_cla=pline->next;

ValueList[l1].Neg.Tra\_lit=ValueList[l1].Neg.Tra\_lit->litline;

}

if (r->first==p) {

r->first=p->nextc;

} else {

for (q=r->first; q!=NULL; q=q->nextc) {//寻找从表头开始第一个只含文字l的单子句

if (q->nextc==p)

break;

}

q->nextc=p->nextc;

}

c=p->sentence;

free(c);

free(p);

free(pline);

r->parasize--;

return OK;

}

/\*恢复认为文字l为真前的CNF邻接链表

参数：(int)文字l:真值为1时求解出错

返回值：OK：邻接链表恢复成功\*/

status RecoverCNF(Root \*r,int l) {

Paradigm \*p;

Clause \*c;

Paraline \*pline;

int l1;

l1=abs(l);

if (l>0) {

//l为正文字，对变元l1的正文字信息链表进行搜索，寻找被删除的子句

for(pline=ValueList[l1].Pos.Tra\_cla; pline; pline=pline->next) {

p=pline->claline;

if(p->flag==l1) {

p->flag=0;

r->parasize++;

for(c=p->sentence; c; c=c->nextl) {

if(c->flag==l1) {

c->flag=0;

p->number++;

}

}

}

}

//l为正文字，对变元l1的负文字信息链表进行搜索，寻找被删除的文字

for (pline=ValueList[l1].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==l1) {

c->flag=0;

p->number++;

}

}

}

} else {

//l为负文字，对变元l1的负文字信息链表进行搜索，寻找被删除的子句

for (pline=ValueList[l1].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->flag==l1) {

p->flag=0;

r->parasize++;

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==l1) {

c->flag=0;

p->number++;

}

}

}

}

//l为负文字，对变元l1的正文字信息链表进行搜索，寻找被删除的文字

for (pline=ValueList[l1].Pos.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==l1) {

c->flag=0;

p->number++;

}

}

}

}

return OK;

}

/\*遍历CNF范式邻接链表\*/

void ParadigmTrabverse(Root \*r) {

Paradigm \*p;

Clause \*c;

int l,value,flag,i=1;

for (p=r->first; p!=NULL; p=p->nextc) {

flag=0;

printf("第%4d行 ",i++);

for (c=p->sentence; c!=NULL; c=c->nextl) {

printf("%4d ",c->literal);

}

printf("\n");

}

printf("遍历完毕！\n");

}

/\*保存CNF范式的解及求解时间信息

参数：(int)solut：CNF公式的解，有解为1，无解为0；

(int)time:DPLL求解时间；

返回值：ERROR：文件打开失败；

OK：解和时间存储成功\*/

status SaveValue(ArgueValue \*ValueList,int solut,int time) {

int i=0;

FILE \*save;

char filenamesave[1000];

for (i=0; filename[i]!='.'; i++)

filenamesave[i]=filename[i];

filenamesave[i++]='.';

filenamesave[i++]='r';

filenamesave[i++]='e';

filenamesave[i++]='s';

filenamesave[i++]='\0';

save=fopen(filenamesave, "wb");

if (save==NULL) {

printf("文件打开失败！\n");

return ERROR;

}

if(solut==1) {

fprintf(save,"s %d\n", solut);

fprintf(save,"v ");

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].IsInit==1) {

if (ValueList[i].Value==1) {

fprintf(save, "%-7d",i);

} else fprintf(save, "%-7d",0-i);

fprintf(save, " ");

}

if(i%10==0) fprintf(save,"\n");

}

fprintf(save, "\nt %d ms\n", time);

}

else if(solut ==0){

fprintf(save,"s %d\n", solut);

fprintf(save,"v ");

fprintf(save, "\nt %d ms\n", time);

}

fclose(save);

return OK;

}

/\*------------------------DPLL算法------------------------\*/

status SAT(void) {

int op=1;//记录选择的操作序号

int choose;//记录选择的算法序号

int i=0,solut=0;//solut记录CNF公式是否有解，有解为1，无解为0

clock\_t start = 0,finish = 0;//记录DPLL函数调用的起始和终止时间

int duration=0;//记录SAT求解时间

FILE \*open;

printf("请输入需求解的cnf文件名：");

scanf("%s",filename);

open=fopen(filename, "r");

if (open==NULL) {

printf("文件打开错误！\n");

getchar();

getchar();

return ERROR;

}

if (CreateParadigm(&open)==ERROR) {//生成CNF公式数据存储结构

getchar();

getchar();

return ERROR;

}

getchar(); getchar();

op=1;

solut=5;

while (op) {

system("cls");

printf("\t\tChoose an option please.\n");

printf("---------------------------------------------------------\n");

printf(" 1.求解该算例 2.遍历CNF文件 3.保存求解结果\n");

printf(" 0.退出\n");

printf("---------------------------------------------------------\n");

scanf("%d",&op);

switch (op) {

case 1:

if (ValueList[1].IsInit==1) {

printf("已对公式求解！");

getchar();

getchar();

break;

}

printf(" 使用：1.优化1算法 2.优化2算法 3.优化前算法\n");

scanf("%d",&choose);

switch (choose) {

case 1:

start=clock();

solut=DPLL1(FindLiteral1(r),1,1);

finish=clock();

duration=(finish-start);

break;

case 2:

start=clock();

solut=DPLL2(FindLiteral2(r),1,1);

finish=clock();

duration=(finish-start);

break;

case 3:

start=clock();

solut=DPLL\_Before(r, 1);

finish=clock();

duration=(finish-start);

break;

default:

printf("输入错误！\n");

break;

}

if (solut) {

AnswerComplete();

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].IsInit==1) {

if (ValueList[i].Value==1) {

printf("%-7d",i);

} else printf("%-7d",0-i);

}

if(i%10==0) printf("\n");

}

} else printf("该公式无解！\n");

printf("\n计算时间为：%d ms\n",(finish-start));

getchar();

getchar();

break;

case 2:

if (solut==5) {

printf("请先求解CNF公式！\n");

getchar();

getchar();

break;

}

ParadigmTrabverse(r);

if (AnswerCheck(solut)==OK) {

printf("公式遍历完成！\n");

} else {

printf("ERROR！\n");

}

getchar();

getchar();

break;

case 3:

if (solut==5) {

printf("请先求解CNF公式！\n");

getchar();

getchar();

break;

}

if (SaveValue(ValueList,solut,duration))

printf("成功存入文件！");

getchar();

getchar();

break;

case 0:

break;

default:

printf("输入错误！");

getchar();

getchar();

break;

}

}

DestroyParadigm(r);//销毁CNF公式数据存储结构

return OK;

}

/\*采用第一种变元选取策略的递归算法DPLL函数

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解\*/

status DPLL1(int num,int op,int timesofDPLL) {

int l,i;

Clause \*c;

Paradigm \*p=r->first;

Paraline \*pline;

i=0;

/\*单子句策略\*/

if (timesofDPLL==1) {

l=0;//第一次进入DPLL函数，不进入单子句循环

} else {

if (op==1)

l=num;

else l=0;

}

while (l!=0) {

stack[++instacksize]=l;

DeleteClause(r, l);

if (DeleteLiteral(r, l)==FALSE) { //删除子句中文字l的反文字

//如探测后公式无解，恢复递归进入本次函数前的邻接链表状态

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

}

if (l>0) {

ValueList[l].Value=1;//真值设置为1

ValueList[l].IsInit=1;//标记已经赋值

} else {

ValueList[0-l].Value=0;//真值设置为0

ValueList[0-l].IsInit=1;//标记已经赋值

}

if (FindLiteral1(r)==0)

return TRUE;//公式中没有剩余的为赋真值的变元，求解成功

l=HasUnitClause(r);//寻找公式中的单子句并将其文字值赋给l

}

if (op==1) {

l=FindLiteral1(r);

} else {

do {

l=rand()%729+1;

for (pline=ValueList[l].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->number!=0)

break;

}

} while (ValueList[l].IsInit==1);

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {

l=c->literal;

break;

}

}

}

if (l==0)

return TRUE;//公式中没有剩余的为赋真值的变元，求解成功

if (DPLL1(l, 1,2)) {

return TRUE;

}

l=0-l;//设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1

if (DPLL1(l,1,2)==FALSE) {

if (instacksize==0)

return FALSE; //l已是本次DPLL1函数第一个进行探测的变元，探测失败

/\*恢复递归进入本次函数前的邻接链表状态\*/

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

} else

return TRUE;

}

/\*采用第二种变元选取策略的递归算法DPLL函数

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解\*/

status DPLL2(int num,int op,int timesofDPLL) {

int l,i;

Clause \*c;

Paradigm \*p=r->first;

Paraline \*pline;

i=0;

/\*单子句策略\*/

if (timesofDPLL==1) {

l=0;

} else {

if (op==1)

l=num;

else l=0;

}

while (l!=0) {

stack[++instacksize]=l;//记录循环过程中真值设为1的文字

DeleteClause(r, l);//删除含文字l的子句

if (DeleteLiteral(r, l)==FALSE) { //删除子句中文字l的反文字

//如探测后公式无解，恢复递归进入本次函数前的邻接链表状态

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

}

if (l>0) {

ValueList[l].Value=1;

ValueList[l].IsInit=1;

} else {

ValueList[0-l].Value=0;

ValueList[0-l].IsInit=1;

}

if (FindLiteral2(r)==0)

return TRUE;

l=HasUnitClause(r);

}

if (op==1) {

l=FindLiteral2(r);

} else {

do {

l=rand()%729+1;

for (pline=ValueList[l].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->number!=0)

break;

}

} while (ValueList[l].IsInit==1);

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {

l=c->literal;

break;

}

}

}

if (l==0)

return TRUE;

if (DPLL2(l, 1,2)) {

return TRUE;

}

l=0-l;//设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1

if (DPLL2(l,1,2)==FALSE) {

if (instacksize==0)

return FALSE;

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

} else

return TRUE;

}

/\*采用第三种变元选取策略的递归算法DPLL函数

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解\*/

status DPLL3(int num,int op,int timesofDPLL) {

int l,i;

Clause \*c;

Paradigm \*p=r->first;

Paraline \*pline;

i=0;

/\*单子句策略\*/

if (timesofDPLL==1) {

l=0;

} else {

if (op==1)

l=num;

else l=0;

}

while (l!=0) {

stack[++instacksize]=l;//记录循环过程中真值设为1的文字

DeleteClause(r, l);//删除含文字l的子句

if (DeleteLiteral(r, l)==FALSE) { //删除子句中文字l的反文字

//如探测后公式无解，恢复递归进入本次函数前的邻接链表状态

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

}

if (l>0) {

ValueList[l].Value=1;

ValueList[l].IsInit=1;

} else {

ValueList[0-l].Value=0;

ValueList[0-l].IsInit=1;

}

if (FindLiteral3(r)==0)

return TRUE;

l=HasUnitClause(r);

}

if (op==1) {

l=FindLiteral3(r);

} else {

do {

l=rand()%729+1;

for (pline=ValueList[l].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->number!=0)

break;

}

} while (ValueList[l].IsInit==1);

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {

l=c->literal;

break;

}

}

}

if (l==0)

return TRUE;

if (DPLL3(l, 1,2)) {

return TRUE;

}

l=0-l;//设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1

if (DPLL3(l,1,2)==FALSE) {

if (instacksize==0)

return FALSE;

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

} else

return TRUE;

}

/\*采用第四种变元选取策略的递归算法DPLL函数

参数：(int)num：该次进入函数真值设为1的变元序号；

(int)op：普通求解cnf文件为1，生成数独终盘为2；

(int)timesofDPLL：第一次进入DPLL函数为1，其后为2；

返回值：FALSE：公式无解；

TRUE：公式有解\*/

status DPLL4(int num,int op,int timesofDPLL) {

int l,i;

Clause \*c;

Paradigm \*p=r->first;

Paraline \*pline;

i=0;

/\*单子句策略\*/

if (timesofDPLL==1) {

l=0;

} else {

if (op==1)

l=num;

else l=0;

}

while (l!=0) {

stack[++instacksize]=l;//记录循环过程中真值设为1的文字

DeleteClause(r, l);//删除含文字l的子句

if (DeleteLiteral(r, l)==FALSE) { //删除子句中文字l的反文字

//如探测后公式无解，恢复递归进入本次函数前的邻接链表状态

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

}

if (l>0) {

ValueList[l].Value=1;

ValueList[l].IsInit=1;

} else {

ValueList[0-l].Value=0;

ValueList[0-l].IsInit=1;

}

if (FindLiteral4(r)==0)

return TRUE;

l=HasUnitClause(r);

}

if (op==1) {

l=FindLiteral4(r);

} else {

do {

l=rand()%729+1;

for (pline=ValueList[l].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->number!=0)

break;

}

} while (ValueList[l].IsInit==1);

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0) {

l=c->literal;

break;

}

}

}

if (l==0)

return TRUE;

if (DPLL4(l, 1,2)) {

return TRUE;

}

l=0-l;//设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1

if (DPLL4(l,1,2)==FALSE) {

if (instacksize==0)

return FALSE;

for (; stack[instacksize]!=num; instacksize--) {

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

}

RecoverCNF(r, stack[instacksize]);

ValueList[abs(stack[instacksize])].IsInit=0;

instacksize--;

return FALSE;

} else

return TRUE;

}

/\*优化前版本DPLL函数

参数：(int)op：普通求解cnf文件为1，生成数独终盘为2；

返回值：FALSE：公式无解；

TRUE：公式有解\*/

status DPLL\_Before(Root \*r,int op) {

int l,i;

Clause \*c;

Paradigm \*p=r->first;

Paraline \*pline;

int \*Memory;

/\*建立Memory栈保存每次DPLL循环过程中真值设为1的文字\*/

Memory=(int\*)malloc((r->litsize)\*sizeof(int));

for (i=0; i<r->litsize; i++)

Memory[i]=0;

i=0;

/\*单子句策略\*/

if (op==1) {

c=HasUnitClause\_Before(r);

} else {

c=NULL;

}

while (c!=NULL) {

l=c->literal;

Memory[i++]=l;

DeleteClause(r, l);//删除含文字l的子句

if (DeleteLiteral(r, l)==FALSE) { //删除字句中文字l的负文字

//如探测后公式无解，恢复递归进入本次函数前的邻接链表状态

if (Memory[0]!=0)

RemoveHeadClaus(r, Memory[0]);//删除增加的单子句

for (i=0; Memory[i]!=0; i++) {

RecoverCNF(r, Memory[i]);

ValueList[abs(Memory[i])].IsInit=0;

}

return FALSE;

}

/\*在变元表中记录变元l的真值\*/

if (l>0) {

ValueList[l].Value=1;

ValueList[l].IsInit=1;

} else {

ValueList[0-l].Value=0;

ValueList[0-l].IsInit=1;

}

if (FindLiteral\_Before(r)==NULL)

return TRUE;//公式中没有剩余的为赋真值的变元，求解成功

c=HasUnitClause\_Before(r);//寻找公式中的单子句并将其文字值赋给l

}

Memory[i]=0;

if (op==1) {

c=FindLiteral\_Before(r);

} else {

do {

l=rand()%729+1;

for (pline=ValueList[l].Neg.Tra\_cla; pline!=NULL; pline=pline->next) {

p=pline->claline;

if (p->number!=0)

break;

}

} while (ValueList[l].IsInit==1);

for (c=p->sentence; c!=NULL; c=c->nextl) {

if (c->flag==0)

break;

}

}

if (c==NULL)

return TRUE;

l=c->literal;

AddClause(r, l);//设文字l真值为1，在CNF范式邻接链表表头增加含文字l的单子句链表

if (DPLL\_Before(r, 1)) {

return TRUE;

}

l=0-l;//设l为真值为1时探测求解失败，故l真值只能为0，即其反文字真值为1

AddClause(r, l);

if (DPLL\_Before(r,1)==FALSE) {

if (Memory[0]!=0)

RemoveHeadClaus(r, Memory[0]);

for (i=0; Memory[i]!=0; i++) {

RecoverCNF(r, Memory[i]);

ValueList[abs(Memory[i])].IsInit=0;

}

return FALSE;

} else

return TRUE;

}

/\*将未赋真值的变元（求解过程中化简舍去）赋真值1\*/

void AnswerComplete(void) {

int i;

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].IsInit==0) {

ValueList[i].IsInit=1;

ValueList[i].Value=1;

}

}

}

/\*检查SAT求解结果正确性

参数：(int)solut：公式求解结果，有解为1，无解为0；

返回值：TRUE：求解正确；

FALSE：求解错误\*/

status AnswerCheck(int solut) {

Paradigm \*p;

Clause \*c;

int flag,l,value,i=0;

if (solut==1) {//公式有解

for (p=r->first; p!=NULL; p=p->nextc) {

i++;

flag=0;

for (c=p->sentence; c!=NULL; c=c->nextl) {

l=abs(c->literal);

if (c->literal>0) {

value=ValueList[l].Value;

} else value=1-ValueList[l].Value;

if (value==1) {

flag=1;//子句中有文字真值为1，子句真值为1

break;

}

}

if (flag==0) {

return FALSE;//子句中无真值为1的文字，子句真值为0，求解错误

}

}

return TRUE;

} else {//公式无解

for (p=r->first; p!=NULL; p=p->nextc) {

flag=0;

for (c=p->sentence; c!=NULL; c=c->nextl) {

l=abs(c->literal);

if (c->literal>0) {

value=ValueList[l].Value;

} else value=1-ValueList[l].Value;

if (value==1) flag=1;//子句中有文字真值为1，子句真值为1

}

if (flag==0) return TRUE;//子句真值为0，求解正确

}

return FALSE;

}

}

/\*------------------------SuDoKu------------------------\*/

status Sudoku(void) {

int op=1,difficulty=0,i,j,flag=1,choose=1;//op和choose记录用户选择的操作，difficulty记录用户选择的数独问题难度

int solut=0;

char c;

int x[3];

srand((unsigned)time(NULL));

while (op) {

system("cls");

printf("\t\tChoose an option please.\n");

printf("---------------------------------------------------------\n");

printf(" 1.数独游戏 2.求解一个数独文件 0.退出\n");

printf("---------------------------------------------------------\n");

scanf("%d",&op);

switch (op) {

case 1:

solut=CreateSudoku();//生成基础数独终盘对应变元真值表

if (solut) {

CNFSudokuTableTransform();//将求解的变元真值表转换为二维数组数独终盘

printf("请选择数独难度：\n1.easy\t\t2.medium\t\t3.Hard\n");

scanf("%d",&difficulty);

//SudokuTablePrint();//测试用句，先输出终盘答案

printf("Loading...");

/\*根据用户选择数独问题难度对数独终盘随机挖相应数量的洞生成问题\*/

switch (difficulty) {

case 1:

DigHole(30);

break;

case 2:

DigHole(50);

break;

case 3:

DigHole(64);

break;

default:

printf("输入出错！\n");

break;

}

choose=1;

while (choose) {

flag=1;//flag标记用户求解答案正确与否，正确为1，错误为0

printf("按“行列值”的顺序，依次输入你的答案（例：“436”代表第四行第三列的空格内填入6），每输入一个答案用回车键断开，输入完成后请输入空格并按回车继续：\n");

getchar();

c=getchar();

while (c!='\n') {

i=0;

while (c!=' ') {

x[i++]=c-'0';

c=getchar();

}

users\_sudoku[x[0]][x[1]]=x[2];//记录用户输入答案

c=getchar();

}

for (i=0; i<9; i++) {

for (j=0; j<9; j++)

//对比用户解答和数独答案，判断其求解正确性

if (sudoku\_table[i][j]!=users\_sudoku[i][j]) {

printf("答案错误！\n");

flag=0;

break;

}

if (flag==0) {

break;

}

}

if (flag) {

printf("恭喜！答案正确！\n");

SudokuTablePrint();//输出数独终盘

choose=0;

} else {

printf("是否查看答案？0.是 1.否\n");

scanf("%d",&choose);

switch (choose) {

case 1:

break;

case 0:

SudokuTablePrint();

break;

default:

printf("输入错误！\n");

break;

}

}

}

} else printf("生成失败！\n");

getchar();

getchar();

break;

case 2:

SolveSudoku();

getchar();

getchar();

break;

case 0:

break;

default:

printf("输入错误！");

getchar();

getchar();

break;

}

}

return OK;

}

/\*创建数独问题转化为SAT问题后的cnf文件\*/

FILE \* CreateSudokuFile(void) {

int x,y,z,i,j,k,l;//x代表数独的行，y代表数独的列，z取1～9分别代表该格中数独填入值为1～9中任一值

FILE \*fp;

fp=fopen("SudokuTableBase.cnf", "wb");

if (fp==NULL) {

printf("文件打开失败！\n");

return NULL;

}

fprintf(fp, "p cnf 729 10287\n");//共有729个变元，9\*9个数独空格每个格对应9个变元，填入1～9中某一值则对应变元为真，其他为假

for (x=0; x<9; x++) {

for (y=0; y<9; y++)

for (z=1; z<9; z++)

for (i=z+1; i<=9; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*x+9\*y+i));

} //每个cell只能取1～9的一个值

for (x=0; x<9; x++) {

for (z=1; z<=9; z++)

for (y=0; y<8; y++)

for (i=y+1; i<=8; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*x+9\*i+z));

} //每行1～9只能各出现一次

for (y=0; y<9; y++) {

for (z=1; z<=9; z++)

for (x=0; x<8; x++)

for (i=x+1; i<=8; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*i+9\*y+z));

} //每列1～9只能各出现一次

for (i=0; i<3; i++) {

for (j=0; j<3; j++)

for (z=1; z<=9; z++)

for (x=0; x<3; x++)

for (y=0; y<3; y++)

for (k=x+1; k<3; k++)

for (l=0; l<3; l++)

if (y!=l)

fprintf(fp, "%d %d 0\n",-(81\*(3\*i+x)+9\*(3\*j+y)+z),-(81\*(3\*i+k)+9\*(3\*j+l)+z));

} //每个region中1～9只能各出现一次

for (x=0; x<9; x++) {

for (y=0; y<9; y++) {

for (z=1; z<=9; z++)

fprintf(fp, "%d ",81\*x+9\*y+z);

fprintf(fp, "0\n");

}

} //每个cell必须取1～9中的一个值

fclose(fp);

fp=fopen("SudokuTableBase.cnf", "r");

return fp;

}

/\*创建新的随机9\*9数独终盘\*/

status CreateSudoku(void) {

int x,y,z,i,d;

int order[82],randnum;

FILE \*fp;

for (x=0; x<9; x++)

for (y=0; y<9; y++)

sudoku\_table[x][y]=0;

do {

fp=CreateSudokuFile();

if (fp==NULL) {

printf("文件打开失败！\n");

return ERROR;

}

CreateParadigm(&fp);//创建数独文件对应CNF范式邻接链表

/\*对1～81的数字进行随机排序\*/

for (i=0; i<=81; i++)

order[i]=i;

for (i=81; i>2; i--) {

randnum=rand()%i+1;

if (i!=randnum) {

d=order[i];

order[i]=order[randnum];

order[randnum]=d;

}

}

for (i=1; i<=10; i++) { //在棋盘中随机选10个格子随机填入1~9

x=(order[i]-1)/9;//顺序为i的cell在棋盘的行数

y=(order[i]-1)%9;//该cell在棋盘的列数

z=rand()%9+1;//1～9的随机数

/\*将已经确定的cell的值记入变元真值表中\*/

for (d=1; d<=9; d++) {

ValueList[81\*x+9\*y+d].IsInit=1;

if (d==z) {

ValueList[81\*x+9\*y+d].Value=1;

DeleteClause(r, (81\*x+9\*y+d));

DeleteLiteral(r, (81\*x+9\*y+d));

} else {

ValueList[81\*x+9\*y+d].Value=0;

DeleteClause(r, -(81\*x+9\*y+d));

DeleteLiteral(r, -(81\*x+9\*y+d));

}

}

}

} while (DPLL2(FindLiteral2(r),2,1)==FALSE);//进入SAT求解器求解，直到得到数独终盘

return OK;

}

/\*求解数独cnf文件并转化为数独终盘形式输出\*/

status SolveSudoku(void) {

FILE \*fp,\*open;

char filename[100];

char read;

int i,j,k,x,y,z,d,cham;

int exitnum[10];

int table[82];

printf("请输入存储数独的文件名：");

scanf("%s",filename);

fp=fopen(filename, "r");

if (fp==NULL) {

printf("文件打开错误！\n");

return ERROR;

}

/\*读取数独文件中的数独问题盘\*/

for (i=0; i<9; i++) {

j=0;

fscanf(fp, "%c",&read);

while (read!='\n') {

if (read=='.') {

sudoku\_table[i][j]=0;

} else if (read<='9'&&read>='1') {

sudoku\_table[i][j]=(int)(read-'0');

} else j=-1;

j++;

fscanf(fp, "%c",&read);

}

}

fclose(fp);

open=SetCNFfile();

if (open==NULL) {

printf("文件打开失败！\n");

return ERROR;

}

CreateParadigm(&open);

i=0;

DPLL\_Before(r,1);//求解数独文件

for (k=1; k<=r->litsize; k++) {

if (ValueList[k].Value==1)

table[i++]=(k%9==0)?9:k%9;

}//将变元表真值信息转换为二维独信息\*/

k=0;

for(i=0; i<9; i++)

for(j=0; j<9; j++)

sudoku\_table[i][j]=table[k++];

printf("数独的解为：\n");

SudokuTablePrint();//输出数独终盘

return OK;

}

//将数独txt文件转化为cnf

FILE \* SetCNFfile(void) {

int i,j,x,y,z,k,l;

printf("输入要转化为CNF文件的文件名：\n");

scanf("%s", filename);

FILE \*fp;

fp = fopen(filename, "w");

if (fp == NULL) {

printf("文件打开失败！\n");

return NULL;

}

int f\_sum = 0;//公式数计数器

for (i = 0; i < 9; i++) {

for(j=0; j<9; j++)

if (sudoku\_table[i][j] != 0) f\_sum++;

}

fprintf(fp,"p cnf 729 %d\n", f\_sum + 10287);

for (i = 0; i < 9; i++) {

for(j=0; j<9; j++)

if (sudoku\_table[i][j] != 0)

fprintf(fp, "%d 0\n", 81\*i+9\*j+sudoku\_table[i][j]);

}

for (x=0; x<9; x++) {

for (y=0; y<9; y++)

for (z=1; z<9; z++)

for (i=z+1; i<=9; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*x+9\*y+i));

} //每个cell只能取1～9的一个值

for (x=0; x<9; x++) {

for (z=1; z<=9; z++)

for (y=0; y<8; y++)

for (i=y+1; i<=8; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*x+9\*i+z));

} //每行1～9只能各出现一次

for (y=0; y<9; y++) {

for (z=1; z<=9; z++)

for (x=0; x<8; x++)

for (i=x+1; i<=8; i++)

fprintf(fp, "%d %d 0\n",-(81\*x+9\*y+z),-(81\*i+9\*y+z));

} //每列1～9只能各出现一次

for (i=0; i<3; i++) {

for (j=0; j<3; j++)

for (z=1; z<=9; z++)

for (x=0; x<3; x++)

for (y=0; y<3; y++)

for (k=x+1; k<3; k++)

for (l=0; l<3; l++)

if (y!=l)

fprintf(fp, "%d %d 0\n",-(81\*(3\*i+x)+9\*(3\*j+y)+z),-(81\*(3\*i+k)+9\*(3\*j+l)+z));

} //每个region中1～9只能各出现一次

for (x=0; x<9; x++) {

for (y=0; y<9; y++) {

for (z=1; z<=9; z++)

fprintf(fp, "%d ",81\*x+9\*y+z);

fprintf(fp, "0\n");

}

} //每个cell必须取1～9中的一个值

fclose(fp);

fp=fopen(filename, "r");

return fp;

}

/\*对已知数独终盘挖num个洞生成有唯一解的数独问题

参数：(int)num：挖洞的数目\*/

status DigHole(int num) {

int x,y,z,i,j,d;

int rank,origin,result=0,circle;

int table\_diged[9][9];

int diged\_cells[82];

FILE \*fp;

for (i=0; i<=81; i++)

diged\_cells[i]=0;

fp=CreateSudokuFile();

if (fp==NULL) {

printf("文件打开失败！\n");

return ERROR;

}

CreateParadigm(&fp);

/\*将数独信息填入对应变元真值表中，并对CNF范式邻接链表作相应处理\*/

for (i=0; i<9; i++)

for (j=0; j<9; j++) {

d=sudoku\_table[i][j];

if (d!=0)

for (z=1; z<=9; z++) {

ValueList[81\*i+9\*j+z].IsInit=1;

if (d==z)

ValueList[81\*i+9\*j+z].Value=1;

else

ValueList[81\*i+9\*j+z].Value=0;

}

}

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].Value==1) {

DeleteClause(r, i);

DeleteLiteral(r, i);

} else {

DeleteClause(r, 0-i);

DeleteLiteral(r, 0-i);

}

}

/\*复制sudoku\_table的值至数组table\_diged\*/

for (i=0; i<9; i++)

for (j=0; j<9; j++)

table\_diged[i][j]=sudoku\_table[i][j];

/\*挖第一个洞\*/

rank=rand()%81+1;

diged\_cells[rank]=1;

x=(rank-1)/9;//顺序为rank的cell在棋盘的行数

y=(rank-1)%9;//该cell在棋盘的列数

origin=sudoku\_table[x][y];

table\_diged[x][y]=0;//记录挖洞的位置

/\*将CNF公式邻接链表恢复至未知第一个洞位置的填入值，即其对应变元真值时的状态\*/

for (z=1; z<=9; z++) {

if (z==origin)

RecoverCNF(r, 81\*x+9\*y+z);

else RecoverCNF(r, -(81\*x+9\*y+z));

ValueList[81\*x+9\*y+z].IsInit=0;

ValueList[81\*x+9\*y+z].Value=0;

}

DestroyParadigm(r);

for (circle=1; circle<num; circle++) {

/\*每次循环重新建立当前数独信息对应CNF公式邻接链表及变元表并对变元表填入相应信息\*/

fp=CreateSudokuFile();

if (fp==NULL) {

printf("文件打开失败！\n");

return ERROR;

}

CreateParadigm(&fp);

for (i=0; i<9; i++)

for (j=0; j<9; j++) {

d=table\_diged[i][j];

if (d!=0)

for (z=1; z<=9; z++) {

ValueList[81\*i+9\*j+z].IsInit=1;

if (d==z)

ValueList[81\*i+9\*j+z].Value=1;

else

ValueList[81\*i+9\*j+z].Value=0;

}

}

for (i=1; i<=r->litsize; i++) {

if (ValueList[i].Value==1) {

DeleteClause(r, i);

DeleteLiteral(r, i);

} else {

DeleteClause(r, 0-i);

DeleteLiteral(r, 0-i);

}

}

/\*挖下一个洞\*/

rank=rand()%81+1;

x=(rank-1)/9;//顺序为rank的cell在棋盘的行数

y=(rank-1)%9;//该cell在棋盘的列数

if (diged\_cells[rank]==1) {

circle--;//该洞已经被挖过

continue;

}

diged\_cells[rank]=1;//记录挖洞的位置

origin=sudoku\_table[x][y];

table\_diged[x][y]=0;

/\*对挖洞位置填入除原终盘数值外的8个数值，依次用SAT求解器进行求解，如有解则说明挖洞后解不唯一，该洞不可挖\*/

for (z=1; z<=9; z++) {

result=0;

if (z!=origin) {

/\*处理CNF范式链表\*/

for (d=1; d<=9; d++) {

ValueList[81\*x+9\*y+d].IsInit=1;

if (d==z) {

ValueList[81\*x+9\*y+d].Value=1;

DeleteClause(r, 81\*x+9\*y+d);

DeleteLiteral(r, 81\*x+9\*y+d);

} else {

ValueList[81\*x+9\*y+d].Value=0;

DeleteClause(r, -(81\*x+9\*y+d));

DeleteLiteral(r, -(81\*x+9\*y+d));

}

}

DPLL1(FindLiteral1(r), 1,1);

SudokuComplete();

result=AnswerCheck(1);

if (result==1) {

table\_diged[x][y]=origin;

break;

} else {

for (d=1; d<=9; d++) {

if (d==z)

RecoverCNF(r, 81\*x+9\*y+d);

else RecoverCNF(r, -(81\*x+9\*y+d));

ValueList[81\*x+9\*y+d].IsInit=0;

ValueList[81\*x+9\*y+d].Value=0;

}

}

}

}

if (result==1) {

circle--;//本次循环挖洞不成功

continue;

}

DestroyParadigm(r);

}

printf("Complete!\n");

/\*输出生成的数独题\*/

printf(" 0 1 2 3 4 5 6 7 8 \n");

printf(" +---+---+---+---+---+---+---+---+---+\n");

for (i=0; i<9; i++) {

printf("%d",i);

for (j=0; j<9; j++) {

if (j==0||j==3||j==6) {

printf("|");

} else printf(" ");

if (table\_diged[i][j]!=0)

printf(" %d ",table\_diged[i][j]);

else

printf(" ");

}

printf("|\n");

if (i==2||i==5||i==8) {

printf(" +---+---+---+---+---+---+---+---+---+\n");

} else printf(" | | | |\n");

}

for (i=0; i<9; i++)

for (j=0; j<9; j++)

users\_sudoku[i][j]=table\_diged[i][j];/\*复制table\_diged的值至数组users\_sudoku\*/

return OK;

}

/\*完善数独终盘\*/

status SudokuComplete(void) {

int i,j,z,y,x,d;

int complete\_table[9][9];

for (i=0; i<9; i++)

for (j=0; j<9; j++)

complete\_table[i][j]=0;

for (i=0; i<9; i++)

for (j=0; j<9; j++)

for (z=1; z<=9; z++)

if (ValueList[81\*i+9\*j+z].Value==1)

complete\_table[i][j]=z;

for (i=0; i<9; i++) {

for (j=0; j<9; j++) {

if (complete\_table[i][j]==0) {

complete\_table[i][j]=1;

for (d=0; d<9; d++)

for (y=0; y<9; y++)//两重循环扫了9遍保证横向一定没有重复

if (y!=j)

if (complete\_table[i][j]==complete\_table[i][y])

complete\_table[i][j]++;

for (d=0; d<9; d++)

for (x=0; x<9; x++)//两重循环扫了9遍保证竖向一定没有重复

if (x!=i)

if (complete\_table[x][j]==complete\_table[i][j])

complete\_table[i][j]++;

}

}

}

for (i=0; i<9; i++)

for (j=0; j<9; j++) {

d=complete\_table[i][j];

if (d!=0)

for (z=1; z<=9; z++) {

ValueList[81\*i+9\*j+z].IsInit=1;

if (d==z)

ValueList[81\*i+9\*j+z].Value=1;

else

ValueList[81\*i+9\*j+z].Value=0;

}

}

return OK;

}

/\*数独对应SAT变元表转化为二维数组\*/

status CNFSudokuTableTransform(void) {

int i,j,z;

for (i=0; i<9; i++) {

for (j=0; j<9; j++) {

for (z=1; z<=9; z++) {

if (ValueList[81\*i+9\*j+z].Value==1) {

sudoku\_table[i][j]=z;

}

}

}

}

return OK;

}

/\*输出数独盘\*/

status SudokuTablePrint(void) {

int i,j;

printf(" 0 1 2 3 4 5 6 7 8 \n");//列标号

printf(" +---+---+---+---+---+---+---+---+---+\n");

for (i=0; i<9; i++) {

printf("%d",i);//行标号

for (j=0; j<9; j++) {

if (j==0||j==3||j==6) {

printf("|");//region间间隔线

} else printf(" ");

printf(" %d ",sudoku\_table[i][j]);

}

printf("|\n");

if (i==2||i==5||i==8) {

printf(" +---+---+---+---+---+---+---+---+---+\n");//region间间隔线

} else printf(" | | | |\n"); //region间间隔线

}

printf("打印完成！\n");

return OK;

}

int main(int argc, const char \* argv[]) {

sum\_op=1;

while (sum\_op) {

system("cls");

printf("\n---Menu For Puzzle Or SAT-----\n");

printf("1.SAT 2.Sudoku\n");

printf("0.Exit\n");

printf("------------------------------\n");

printf("请输入0~2：\n");

printf("----------\n");

scanf("%d",&sum\_op);

switch (sum\_op) {

case 1:

SAT();

break;

case 2:

Sudoku();

break;

case 0:

break;

default:

printf("输入错误！");

getchar();getchar();

break;

}

}

return 0;

}