

**课程设计报告**

**题目：基于SAT的二进制数独游戏求解程序**

**课程名称：程序设计综合课程设计**

**专业班级： CS1806**

**学 号： U201814788**

**姓 名： 刘美**

**指导教师： 纪俊文**

**报告日期： 2020/05/07**

**计算机科学与技术学院**

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：**将二进制数独游戏[5，6]问题转化为SAT问题[6]，并集成到上面的求解器进行问题求解，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[6-9]。(15%)

目录

[**设计内容 II**](#_Toc25415)

[**设计要求 II**](#_Toc30705)

[**1.1 课题背景与意义 4**](#_Toc12989)

[**1.2 国内外研究现状 4**](#_Toc32427)

[**1.3 课程设计的主要研究工作 4**](#_Toc12763)

[**2.1 系统需求分析 5**](#_Toc28561)

[**2.2 系统总体设计 5**](#_Toc7187)

[**3.1 有关常量，结构体和函数定义 5**](#_Toc28214)

[**3.2 一些函数的设计 11**](#_Toc5845)

[**3.3 DPLL算法思想 14**](#_Toc16259)

[**3.4 DPLL算法优化思想 15**](#_Toc8844)

[**3.5 数独 18**](#_Toc21542)

[**4.1 系统实现 21**](#_Toc6504)

[**4.2 系统测试 21**](#_Toc15583)

[**5.1总结 45**](#_Toc13432)

1. 引言
   1. 课题背景与意义

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。SAT问题也是程序设计与竞赛的经典问题。

由于SAT是NP类问题，如果他能得到高效解决，那么一定可以高效地解决所有其他的NP类问题，因为所有的NP类问题都能在多项式时间内进行相互转化，所以设计和实现SAT问题高效的求解算大意义重大。

* 1. 国内外研究现状

目前SAT算法大致可以归结为两大类：完备算法（回溯搜索算法）和局部搜索算法。完备算法基于穷举，其优点是保证找到对应SAT问题的解或证明公式不满足，但是效率极低。局部算法由于采用了启发式策略来指导搜索，使得求解速度相对较快，但是在某些实例上可能得不到解，即不能保证SAT问题的不可满足性。

最经典的求解SAT的完备算法是DPLL算法，也是本次实验的基础算法。它是由Davis和Putnam等人在1960年提出的，其余的完备算法大都是在DPLL算法的基础上衍生出来的。后来又涌现出了许多高效的SAT算法如MINISAT,SATO,CHAFF,POSIT和GRASP等。国内也涌现出了许多高效的求解算法，如梁东敏提出的加权WAST算法，金人超和黄文奇提出的并行Solar算法等。

尽管SAT算法以及取得了举足轻重的改进，但有些问题还是没有得到高效的解决，因此研究并实现高效率的求解算大仍是当前解决的重心问题之一。

* 1. 课程设计的主要研究工作

1. 理解并实现DPLL算法，并基于DPLL算法求解CNF文件
2. 在DPLL的基础上进行优化，降低运行时间和消耗
3. 实现SAT问题的求解后，运用到二进制数独，实现二进制数独的求解，设计求解二进制数独的游戏程序并具有一定的交互性
4. 系统需求分析
   1. 系统需求分析
5. 能够读取cnf文件并进行解析，在利用DPLL算法或基于DPLL算法的优化算法进行求解
6. 由用户输入二进制数独，将二进制数独游戏转化为SAT问题，并集成到上面的求解器进行问题求解。

1. 系统总体设计

3.1系统总体设计



图3-1 系统总体设计

3.2 有关常量，结构体和函数定义

（1）常量

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE -1

#define NOTSURE 0

#define NOTCONTAIN 2

#define EVALUE(x) ((x>0)?1:-1)

#define CLAUSE 0

#define LITERAL 1

#define SPLIT 2

#define DEBUG2 0

#define VSIDS 2

#define VSIDSCOUNT 100

#define LEARNLENGTH\_MAX 20 //限制学习子句的数目

#define OUTTIME 120000

（2）有关结构体

typedef int status;

typedef char boolean;

（3）文字变量结构体

typedef struct Node{

int var; //正为正，负取非

struct Node\* next; //指向下一个变元

}Node; //变元定义

（4）子句

typedef struct Clause{

int length; //子句中变元的数目

Node\* head; //指向子句中的第一个变元

Node\* rmv; //指向第一个被移除的变元

status isremoved; //子句是否被移除子句集

struct Clause\* pre; //指向上一条子句

struct Clause\* next; //指向下一条子句

}Clause; //子句定义

（5）CNF

typedef struct CNF{

int varnum; //变元个数

int clausenum; //子句的数目

int\* varfloor; //变元所在的决策层

int\* var; //变元取值数组，1为真，0为假，-1为不确定

int\* vsids; //VSIDS策略积分器数组

Clause\* root; //指向第一个子句

struct LearnClause\* learn\_root; //指向第一个学习子句

struct LiteralIndex\* lindex;//变元索引,指向第一个索引

}CNF; //cnf子句集定义

（6）学习子句

typedef struct LearnClause{

boolean isInStack; //判断是否在回溯栈中

int floor; //决策层数

int count; //调用次数

Clause\* clause; //指向该学习子句对应的子句

struct LearnClause\* next; //指向下一条学习子句

}LearnClause;

（7）索引

typedef struct LiteralIndex { //cnf索引的next指向子句索引，下标对应变元

Node\* Np; //指向与下标一致的变元

Clause\* Cp; //指向变元出现的子句

struct LiteralIndex\* next;//指向下一个索引

} LiteralIndex;

（8）回溯栈

typedef struct Stack{

int tag; //标识子句或变元

int floor; //决策层数

Node\* Np;

Clause\* Cp;

struct Stack\* next; //下一个修改区

}ChangeStack; //保存修改

（9）回溯层

typedef struct Floor{

int a; //冲突变元

int floor; //变元所在的决策层

}Floor;

（10）数独

typedef struct Sudoku{

int sdk[6][6];

struct Sudoku\* next;

}Sudoku;

（11）相关函数

status createCNF(); //创建CNF

status destroyCNF(); //销毁CNF

status clearCNF(); //清空cnf答案

status createLIndex(); //创建变元索引

status addLIndex(); //添加索引

status deleteClauseLIndex(); //删除子句变元索引

status deleteallLIdex(); //删除所有变元索引

status addClause(); //添加子句

status insertClause(CNF\* cnf,Clause\* cl); //顶部插入子句

Clause\* removeClause(); //移除子句

status deleteClause(); //删除子句

status deleteallClause(); //删除所有子句

status deleteLearnClause(); //删除学习子句

status sortvar(); //子句变量排序

status arrayAssign(); //变元真假取值保存到数组

boolean isClauseEmpty(); //子句是否为空

boolean isClauseUnit(); //判断是否为单元子句

boolean evaluateClause(); //子句表达式是否为真

boolean haveEmptyClause(); //是否有空子句

boolean haveClause(); //子句集是否为空

Clause\* LocateUnitClause(); //是否有单元子句并返回

status printClause(); //打印子句

status printLearnClause(); //打印学习子句

status printLIndex(); //打印索引

status addvar(); //添加变元

status deletevar(); //删除变元

status backvar(); //恢复变元

Node\* removevar(); //移除变元，并返回

status printcnf();

status DPLL();

status advancedDPLL();

status SimplifySingleClause(); //化简单子句

status MOM(CNF\* cnf);

status VSIDSS(CNF\* cnf);

status combineStrategy();

status saveChange(); //保存修改

status backChange(); //恢复修改

status createLearnClause(CNF\* cnf,int\* a,int i,int x); //生成学习子句

status backLearnClause(CNF\* cnf,int floor); //学习子句回溯

status deleteRepeateLearnClause(); //删除重复的学习子句

status backAssign(CNF\* cnf,int floor); //赋值回溯

status compare\_Des(const void\* a,const void\* b);

void CNFpage();

void SUDOKUpage();

CNF\* loadCNF(); //读取cnf文件

status changetoRes(char\* path,char\* cnfpath ); //求解结果输出到同名res文件

status saveRes(char\* respath,status dpll,CNF\* cnf,int time); //保存res文件

status saveCNF(); //保存cnf文件

Sudoku\* createSudoku();

CNF\* transtoCNF(); //转换为cnf文件

status solveSudoku(); //求解数独

status DFSSSudoku();

status printSudoku(); //打印数独

Sudoku\* transtoSudoku(); //cnf转换为数独

status randSudoku(); //随机生成数独

status rule2\_select4unit(); //约束2，找任意四个单元

//添加到cnf文件中时子句转换的约束

status rule1();

status rule2();

status rule3();

//生成数独时的约束

status checkRule1();

status checkRule2();

status checkRule3();

status checkSudoku(); //生成数独填数是检查是否满足约束

status SaveSudoku(); //保存输入的数独到文件中

Sudoku\* loadSudoku(); //读取文件中的数独文件

status SolveMySukodu(); //验证输入的数独是否满足数独约束规则

status istheSameSudoku(); //检查输入的数独是否是待求解的数独

该实验的存储结构为文字邻接表，数据类型有子句集，子句，文字，学习子句，索引，变元，修改栈，冲突变元与决策层和数独。其中，子句集（CNF）中存储了cnf文件的子句数目和变元总数，并含有指向子句，学习子句，变元索引的指针，以及变元取值数组，决策层数组，VSIDS决策数组。子句结构中存储了子句的长度和该子句是否被移除，指向第一个变元和第一个移除变元，前一条子句和下一条子句的指针。变元结构中存储了变元和指向下一个变元的指针。学习子句，即产生冲突的字句，结构中存储了是否在回溯栈中，决策层数，调用次数，以及指向同子句和下一条学习子句的指针。索引结构体中包括指向该变元所在的文字结构和子句结构的指针，以及指向下一条索引的指针。

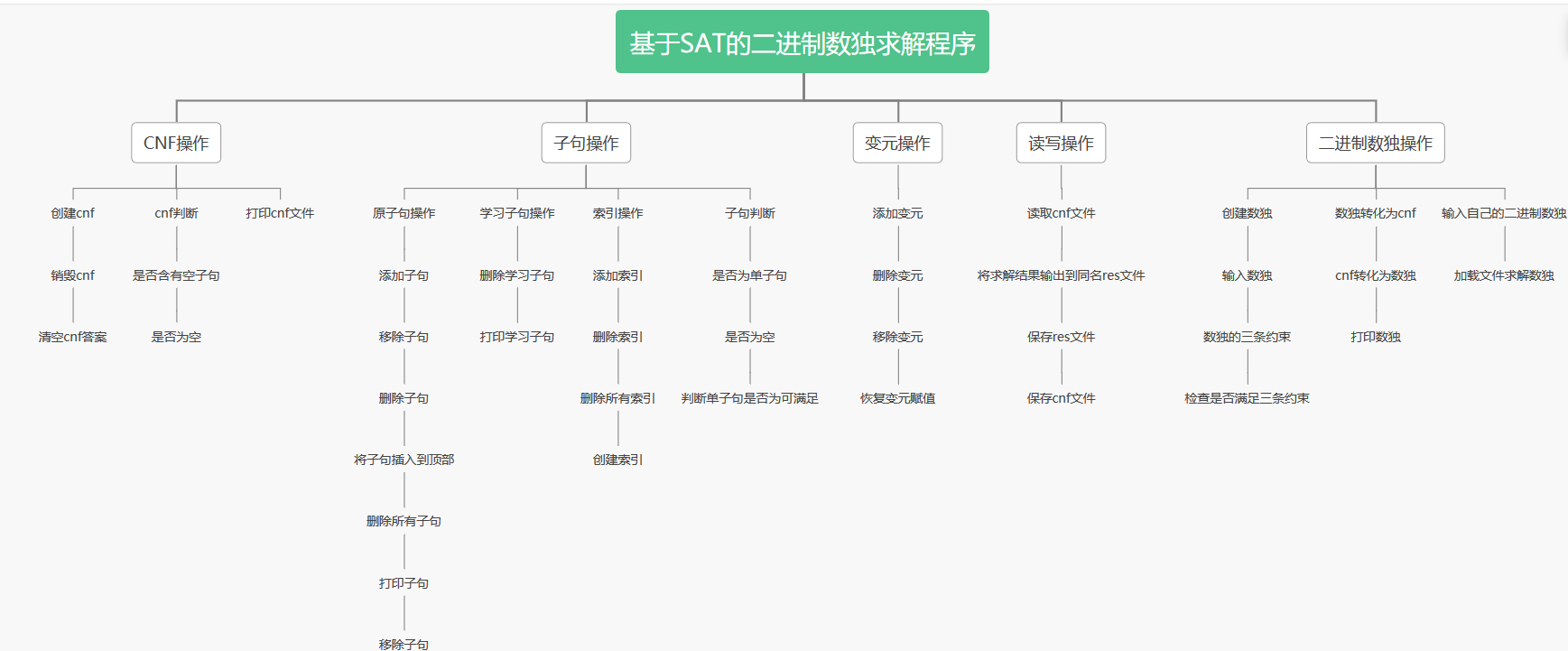


图3-1 系统操作设计

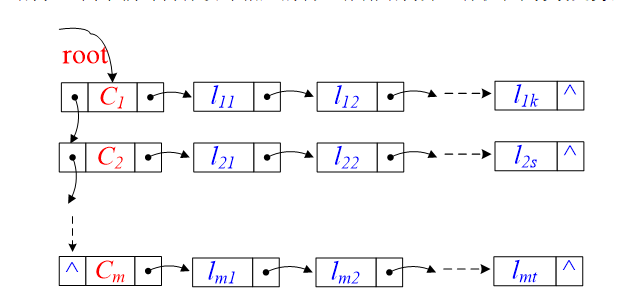


图3-2 cnf公式存储结构图

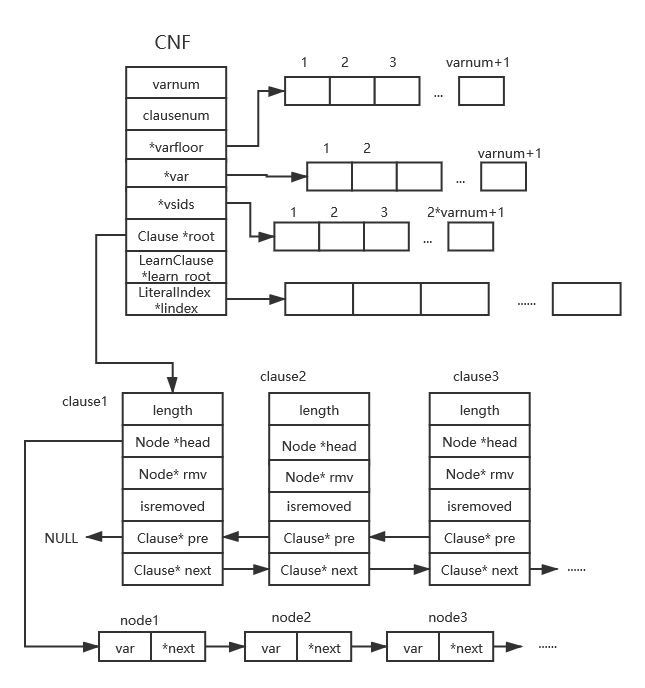


图3-3 cnf结构示意图

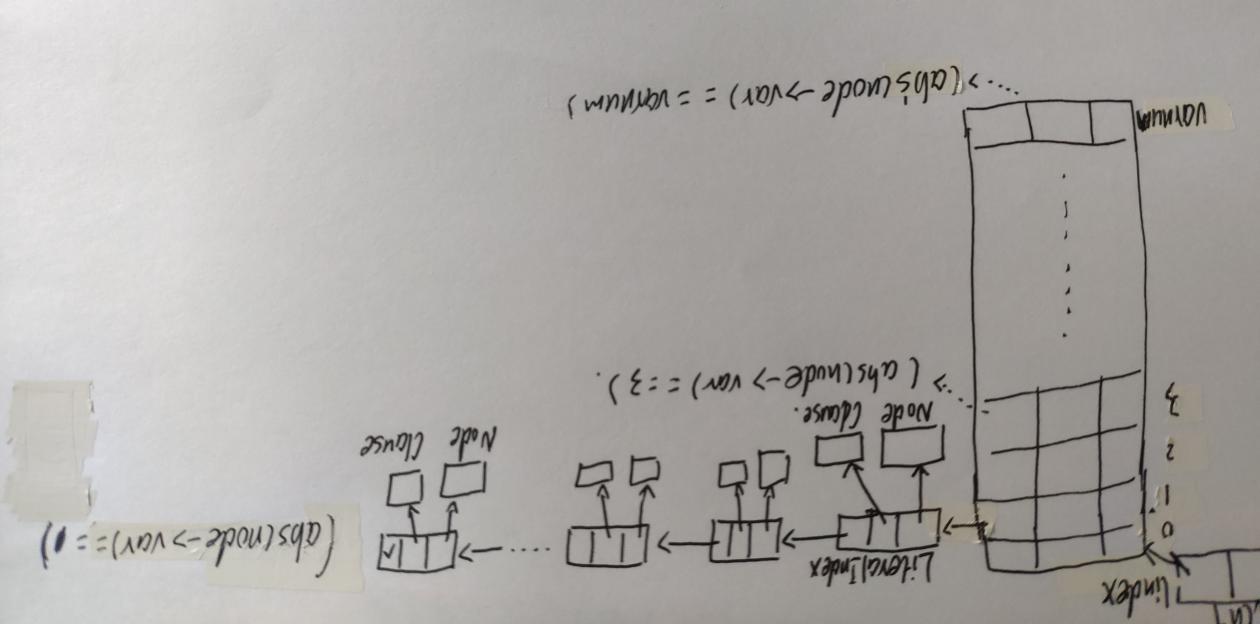


图3-4 索引结构图

3.3 一些函数的设计

1. 函数名称：status SimplifySingleClause(CNF\* cnf,int var,ChangeStack\* head)

初始条件:传入的变元var赋值为真

操作结果：简化单子句

算法设计：

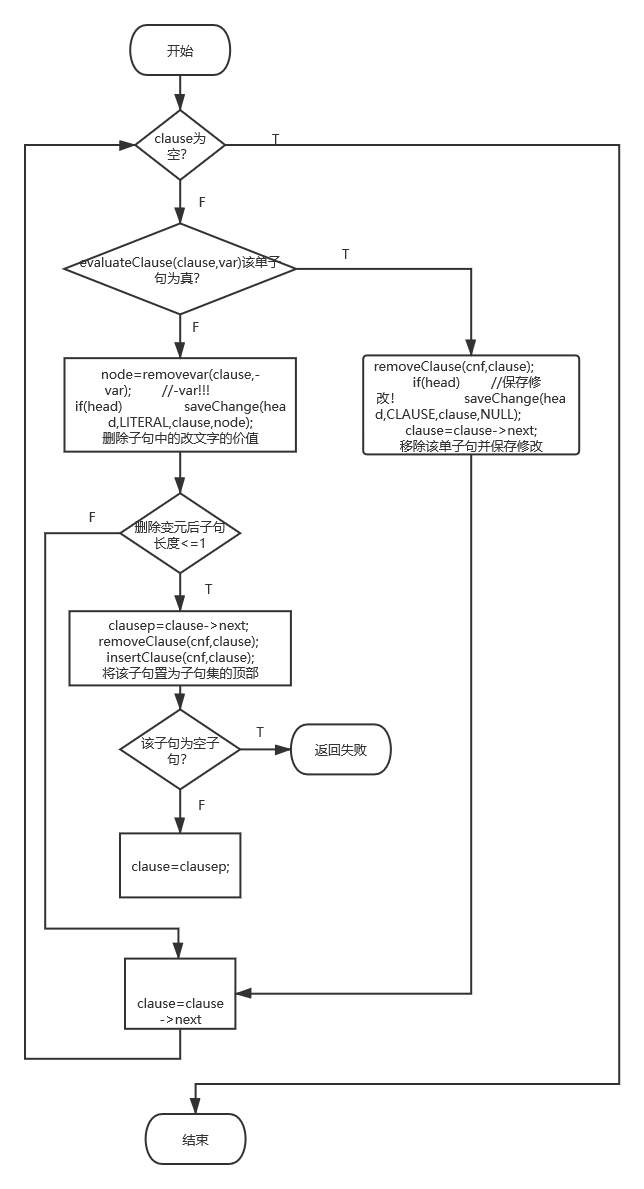


图3-5 单子句化简算法流程

1. 函数名称：status MOM(CNF\* cnf)

操作结果：利用MOM算法选出决策变量

算法设计：

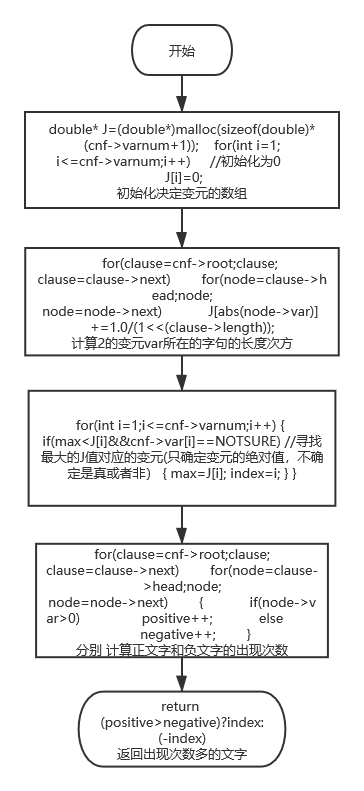


图3-6 MOM算法流程

3.4DPLL算法思想

DPLL算法是基于树/二叉树的回溯搜索算法，主要使用两种基本处理策略：

单子句规则。如果子句集*S*中有一个单子句*L*,那么*L*一定取真值，于是可以从*S*中删除所有包含*L*的子句（包括单子句本身），得到子句集*S*1，如果它是空集，则*S*可满足。否则对*S*1中的每个子句，如果它包含文字*¬L*,则从该子句中去掉这个文字，这样可得到子句集合*S*2。*S*可满足当且仅当*S*2可满足。单子句传播策略就是反复利用单子句规则化简*S*的过程。

分裂策略。按某种策略选取一个文字*L*.如果*L*取真值，则根据单子句传播策略，可将*S*化成*S*2；若*L*取假值（即*¬L*成立）时，*S*可化成*S*1。

其算法可大致表示为：

DPLL( S) :

/\* S为公式对应的子句集。若其满足，返回TURE；否则返回FALSE. \*/

{

while(S中存在单子句) {

//单子句传播在S中选一个单子句L；依据单子句规则，利用L化简S；

if S = Φ return(TRUE);

else if (S中有空子句 ) return（FALSE）；

}//while

基于某种策略选取变元v； //策略对DPLL性能影响很大

if DPLL（S ∪v ）return(TURE); //在第一分支中搜索

return DPLL(S ∪¬v); //回溯到对v执行分支策略的初态进入另一分支

}

选取变元v的思想采用的最短子句出现频率最大优先（MOM）方法。也就是常说的JW方法。该算法的重点是求出和文字l相对于的J值，其所对应的求解公式如下：



简单来说，一个子句的长度越大，使该子句可满足的“候选文字”数目越多，相对来说就越容易满足，其J值同时则越小。所以首先找出J值最小对应的子句，然后，统计该子句中所含有的出现频率最大的文字，该文字即为所选的决策变量。

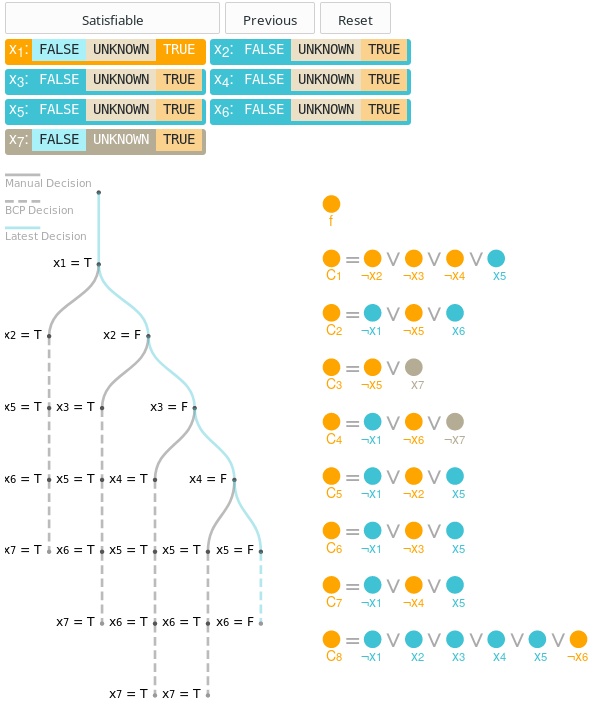


图3-7 DPLL算法实现过程示例

3.5 DPLL算法优化思想

优化的DPLL算法（CDCL算法），改变了决策变量选取的方式，改为了VSIDS策略。

VSIDS策略即为每一个变量的正负文字分别设置一个计数器，并设为该文字在所有子句集中出现的次数。若文字在某一子句（包括学习子句）中出现一次则自增1，而一旦发生冲突，就将冲突导出子句添加到子句集中，并增加冲突导出子句中相应的文字计数器。最后在按照未赋值变量中的文字计数器的排序中选择最大的一个文字作为决策变量。同时为了防止一些变量长时间得不到赋值，经过一次决策次数（LEARNLENGTH\_MAX =20），每一个文字的计数器的值均除以2。

要达到优化，还需要借助于冲突分析、子句学习和回溯机制。本次实验所采用的回溯机制为非时序回溯，即回溯时不仅仅是简单的回到上一决策层，而是可能回跳多个决策层。理论上的返回值应该首先找到所谓蕴含图中的唯一蕴含点以及与冲突相关的变元所在的决策层，再从这些决策层中选出最大的那个就是应返回的层数。

为实现子句学习和非时序回溯，实验中创建了一个栈来保存进行单子句简化时的修改，（因为栈是后进先出，最近保存的修改即最近的赋值最先恢复）在运行优化的dpll算法时，如果检测到出现空子句，意味着发生了冲突，此时恢复其原来的赋值，同时将这个发生冲突的变元加入到学习数组a[]\_0中。将学习数组定义为第0号元素为数组的长度，1-i为冲突的变元，第i+1个位唯一 蕴含点。第i次撤销后，先检查该子句是否有元素，若只有一个元素且标志为子句同时该元素在学习数组a[]\_0中（意味着与冲突子句有关），则将该子句的元素全部加入到数组a[]\_0中。当所有撤销完成后，此时学习数组记为a[]\_1,删去a[]\_1中本层所有赋值的变元得到a[]\_2，此时a[]\_2中出现的决策层最大的值返回的决策层。将本层的决策加入到a[]中，即得到学习子句。

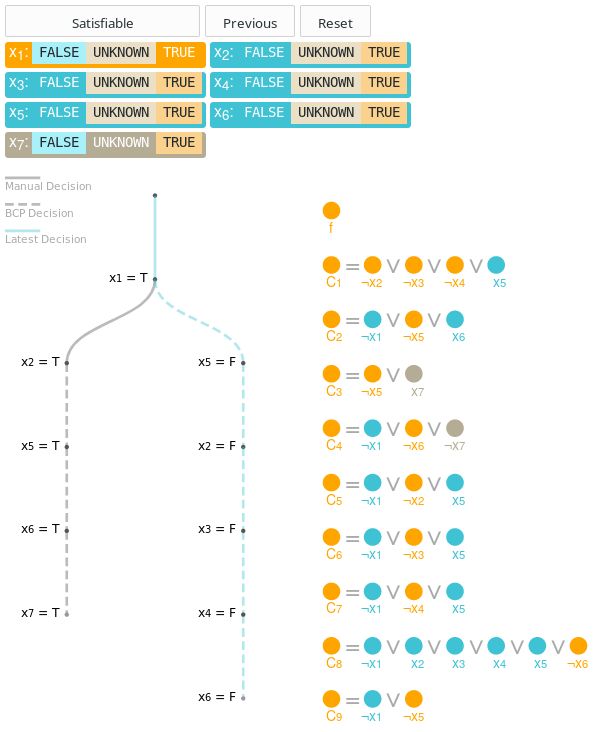


图3-8 DPLL优化后的运行示例

在为x2赋值为真后由BCP过程推测出x5=T,x6=T，x7=T，此时得到空子句，回溯，可以由第二个分支看出，此时的回溯并不是简单的回溯到上一层（x2=T），而是通过学习子句进行BCP得到x5=F，进而继续进行BCP过程得到满足结果。

此外还有子句删除，这主要是为了防止无限制的子句学习消耗内存，所以限制学习子句的数目限制并考虑学习子句的先后顺序。

具体的优化解释见源代码注释。

无论是优化前或者优化后的DPLL算法都采用的递归思想。

3.6 数独

数独部分实现了用户输入自己的数独并保存在文件中、读取用户保存的数独盘文件并进行求解和随机生成一个数独由用户进行求解或者系统求解的功能。

数独的读写操作和cnf的读写操作类似，不做赘述。

数独的生成采用挖洞法生成，并实现了生成的数独盘的唯一性。

为了实现程序的交互性，设计可以用户自己求解系统随机生成的数独也可直接系统求解。自己求解失败后可继续求解或者直接系统求解。为了正确验证用户输入的数独是否成功求解，不仅需要输入的数独满足数独的三条约束规则，还要保证输入的数独是我随机生成的数独而不是其他满足规则的二进制数独。对于检验是否是同一个数独可以利用两个for循环比较原始数独盘中非-1单元格的数值是否和输入数独盘中对应的值相同，若相同则视为是同一个数独，否则输入错误的数独。

生成数独盘算法流程如下：

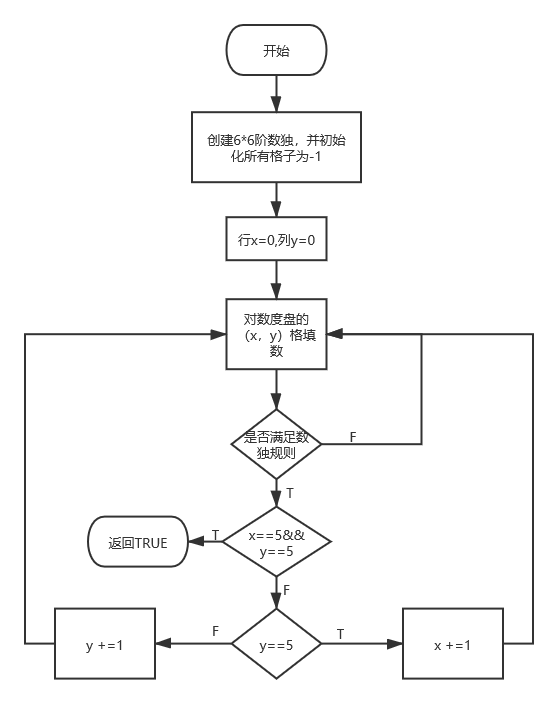


图3-9 随机生成数独算法流程

确定数独盘的唯一性算法流程如下：

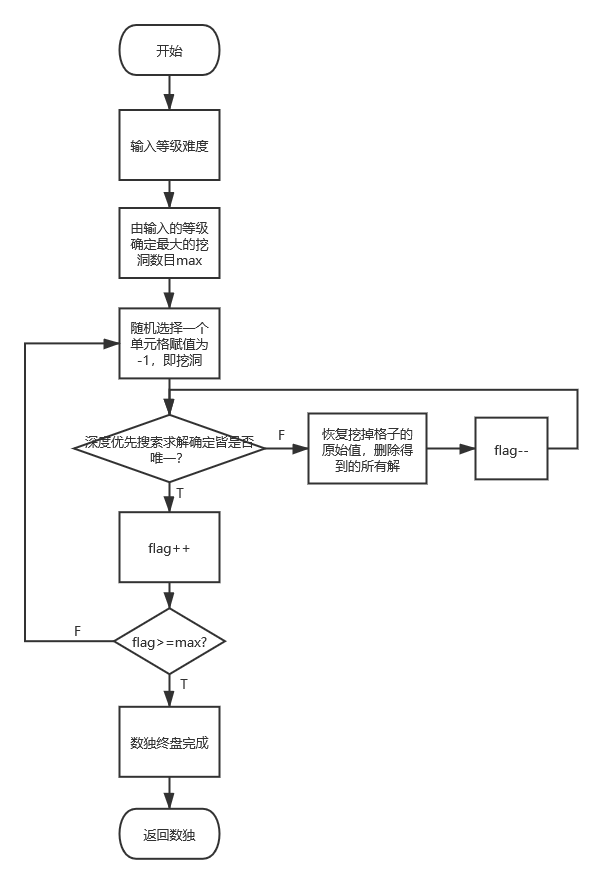


图3-10 确定数独唯一性算法流程图

深度搜索确定是否存在多解，如果存在多个解，保存所有解。采用穷举搜索的方法对数独盘从上至下，从左至右，找到待填格进行填数尝试，即填入0-1的数字（0、1均要检测），同时检验填入的数字是否满足数独游戏的规则。若能满足则视此次填入的数字可行。若在一个空格尝试中，0-1均无法填入，则回溯到前一个所尝试的空格中并将其已经填入的数字换成另外一个数字。如此往复直至找到所有的解。

调用深度搜索探测解的个数之后，若得到多解，则恢复当前挖掉格的赋值（意味着这个格子不能挖），并且释放掉得到的解的空间。否则的话，继续随机生成一个空格，重复求解--检测解的唯一性的步骤直至挖掉的格子的数目满足自己设置的要求。

1. 系统测试
   1. 系统实现

本设计实现了基于DPLL过程实现一个高效SAT求解器，对于给定的中小规模算例进行求解，输出求解结果，统计求解时间。同时将该求解器运用到二进制数独的求解程序。

本次实验中使用的环境配置如下：

操作系统版本：Windows 10 专业版

编译器及其版本：GNN GCC Compiler

编程环境：CodeBlocks ver17.12

* 1. 系统测试

4.2.1 界面展示

用户主界面：

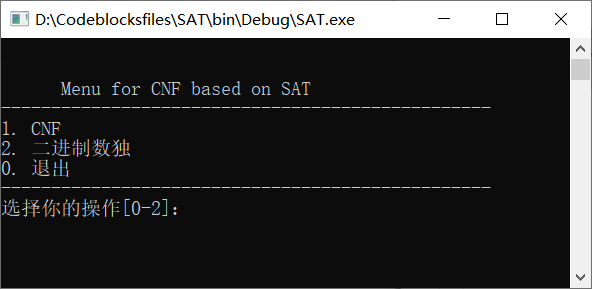


图4-1 用户主界面

CNF界面：

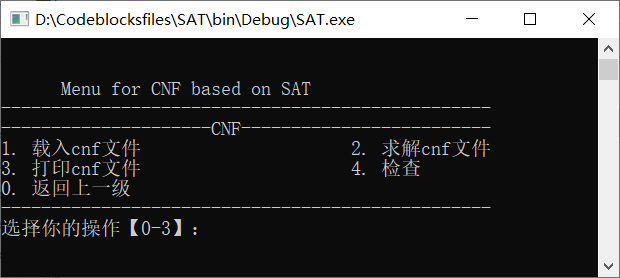


图4-2 CNF界面

数独界面：

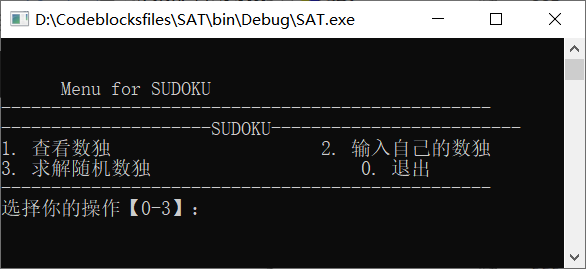


图4-3 数独界面

4.2.2 CNF算例求解

表4-1 S型算例

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变元数 | 子句数 | 子句数与变元数的比值 | 算例名称 | 可满足性 | 优化后 | 优化前 | 优化率 |
| 25 | 100 | 4 | tst\_v25\_c100.cnf | 可满足 | 1ms | 1ms | 0 |
| 100 | 600 | 6 | problem11-100.cnf | 可满足 | 2ms | 4ms | 50% |
| 100 | 200 | 2 | problem9-100.cnf | 可满足 | 0ms | 2ms | 100% |
| 100 | 367 | 3.67 | problem3-100.cnf | 可满足 | 1ms | 5ms | 80.00% |
| 20 | 1783 | 89.15 | 7cnf20\_90000\_90000\_7.shuffled-20.cnf | 可满足 | 23ms | 25ms | 8.00% |

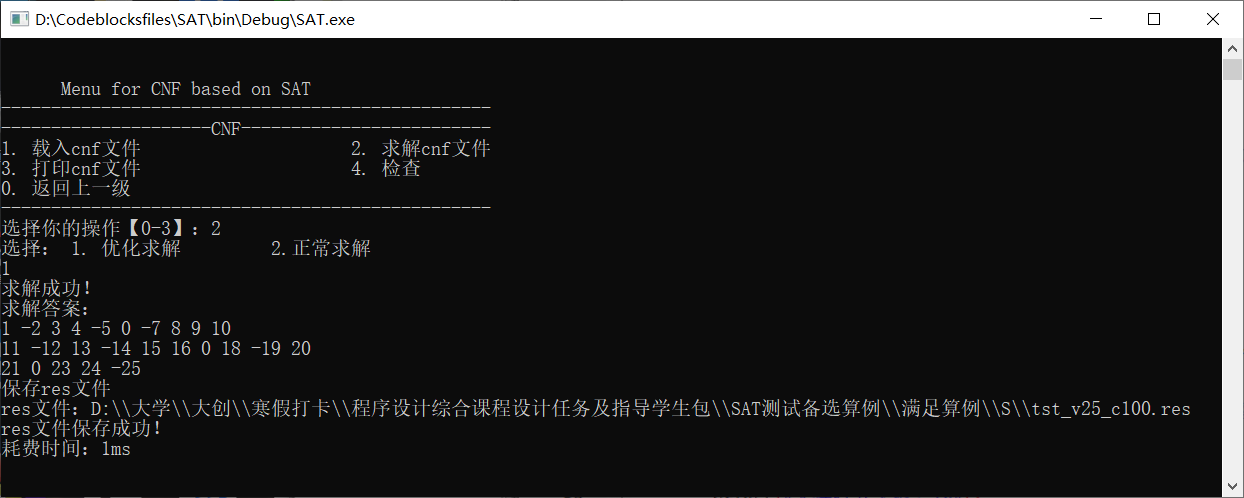


图4-4 小型算例1优化后结果

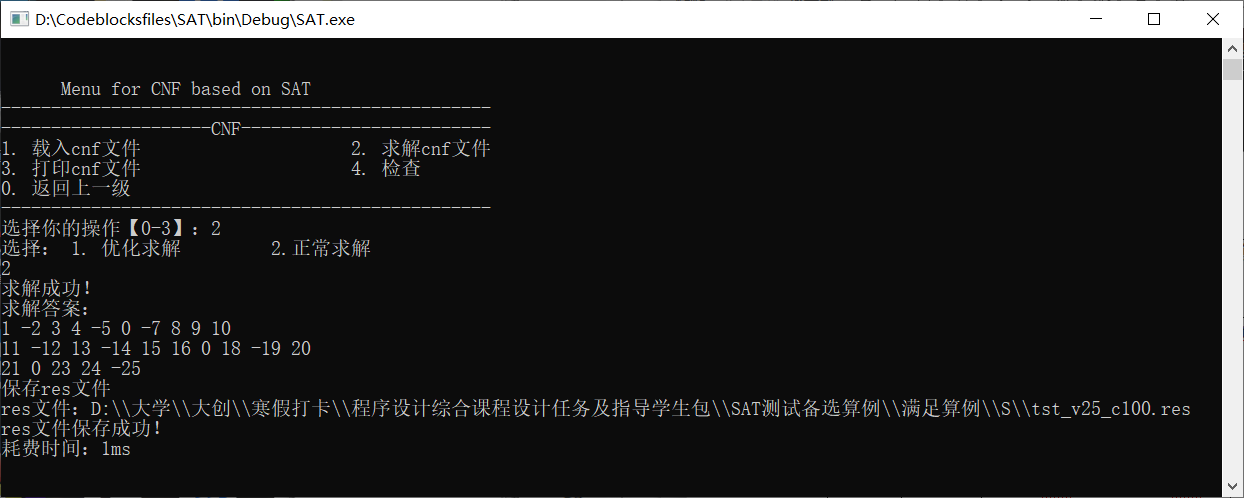


图4-5 小型算例1优化前结果

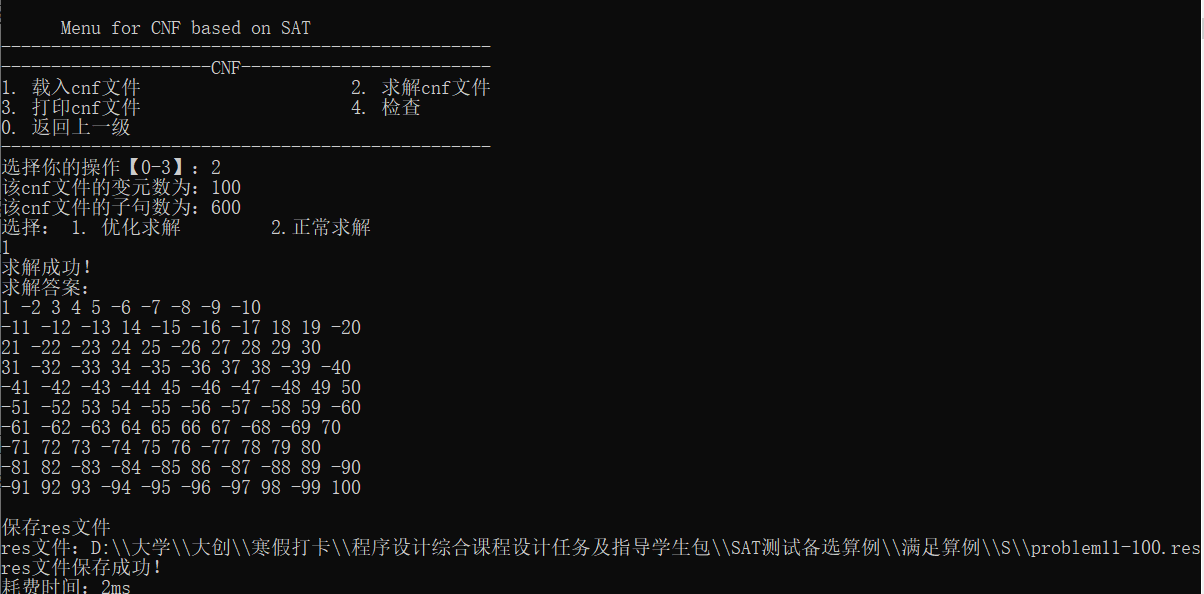


图4-6 小型算例2优化后结果

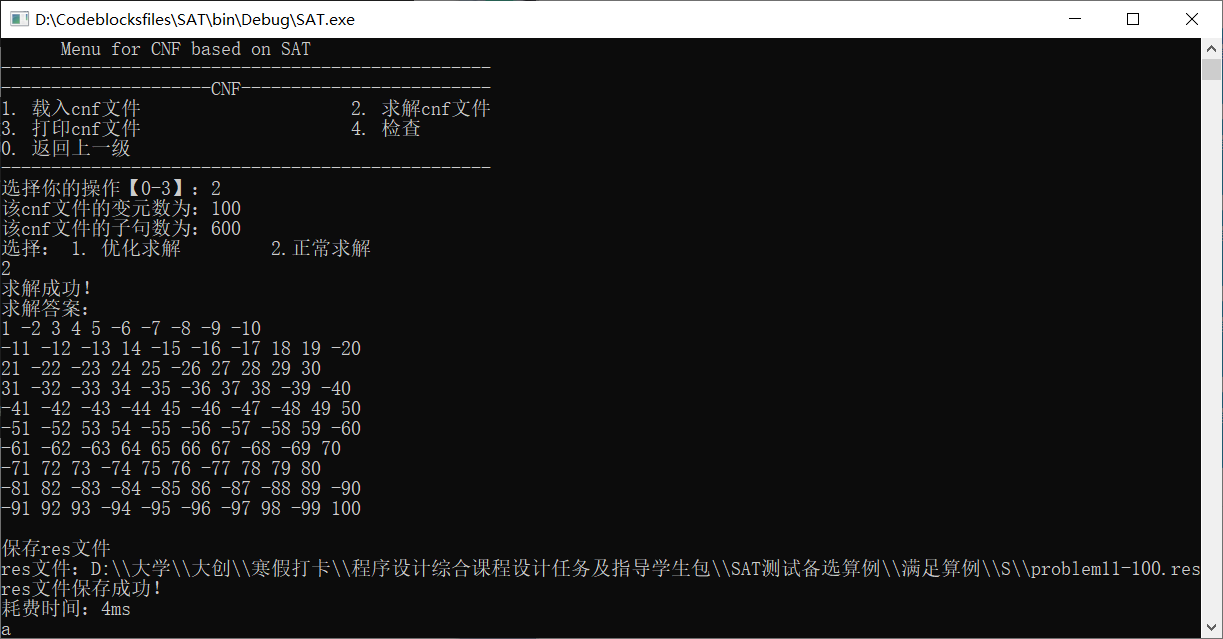


图4-7 小型算例2优化前结果

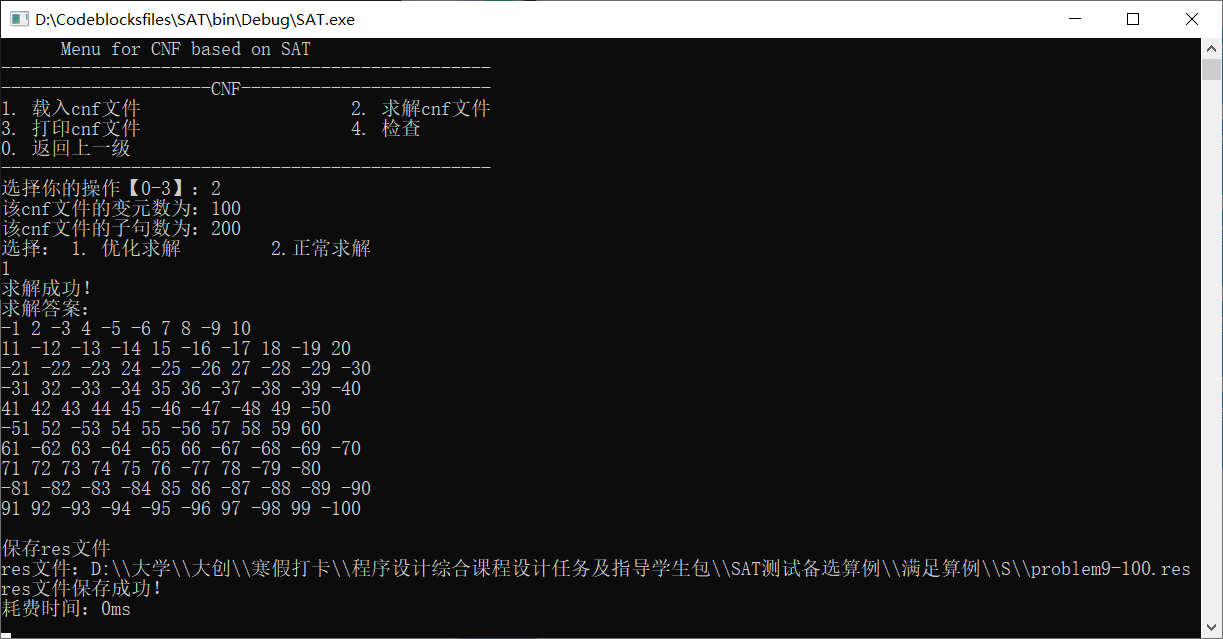


图4-8 小型算例3优化后结果

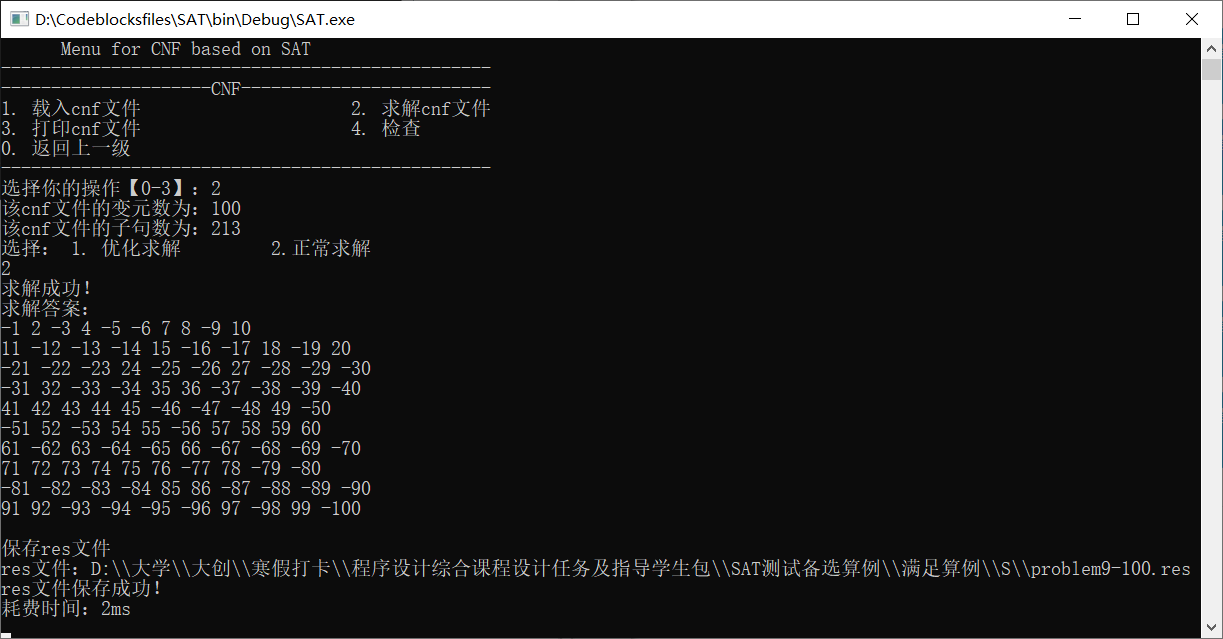


图4-9 小型算例3优化前结果

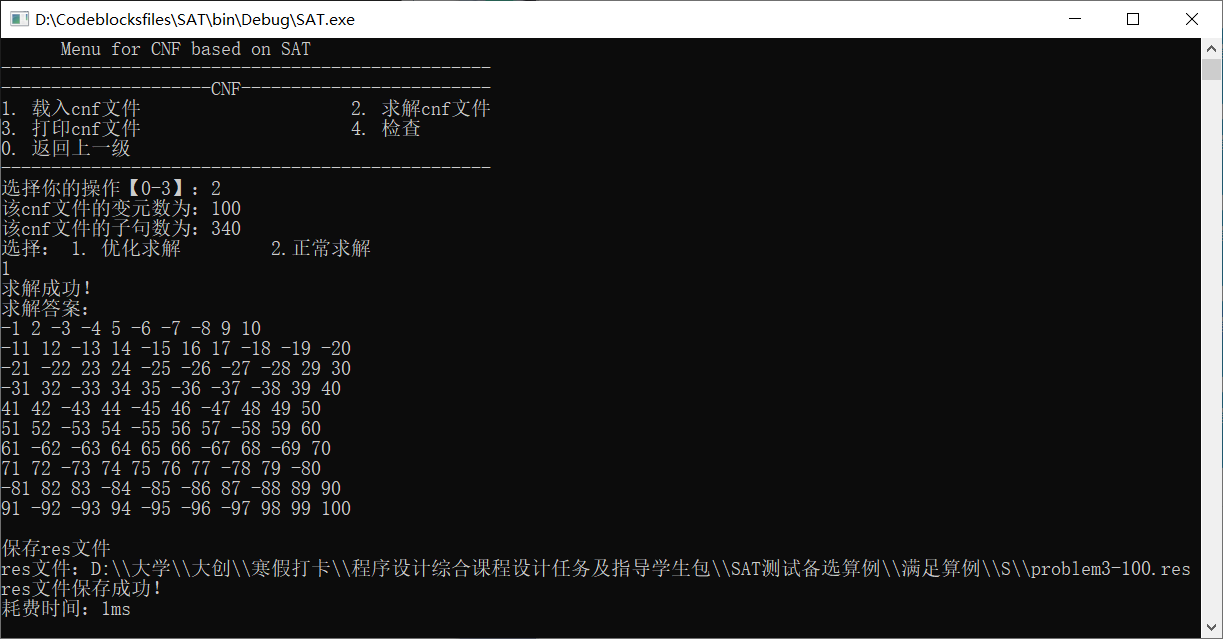


图4-10 小型算例4优化后结果

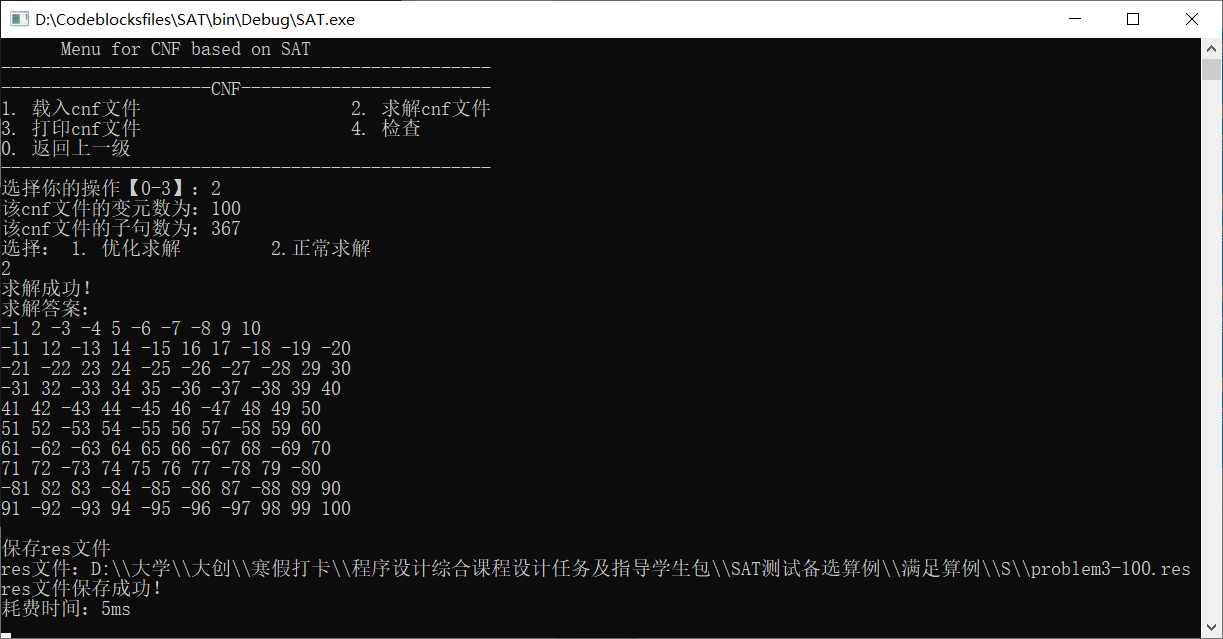


图4-11 小型算例4优化前结果



图4-12 小型算例5 优化后结果



图4-13 小型算例5优化前结果

表4-2M型算例：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变元数 | 子句数 | 子句数与变元数的比值 | 文件名称 | 可满足性 | 优化后 | 优化前 | 优化率 |
| 231 | 1166 | 5.05 | bart17.shuffled-231.cnf | 可满足 | 414ms | 82417  ms | 99.50% |
| 354 | 2596 | 7.33 | ec-mod2c-rand3bip-sat-250-2.shuffled-as.sat05-2534.cnf | 不确定 | 240001ms | 240086ms | 不确定 |
| 224 | 1762 | 7.87 | sud00082.cnf | 可满足 | 15ms | 1348ms | 98.89% |
| 232 | 1901 | 8.19 | sud00012.cnf | 可满足 | 42ms | 496ms | 91.53% |
| 301 | 2780 | 9.24 | sud00001.cnf | 可满足 | 44ms | 22762  ms | 99.81% |
| 303 | 2851 | 9.41 | sud00009.cnf | 可满足 | 16ms | 2274ms | 99.29% |



图4-14中型算例1优化后结果

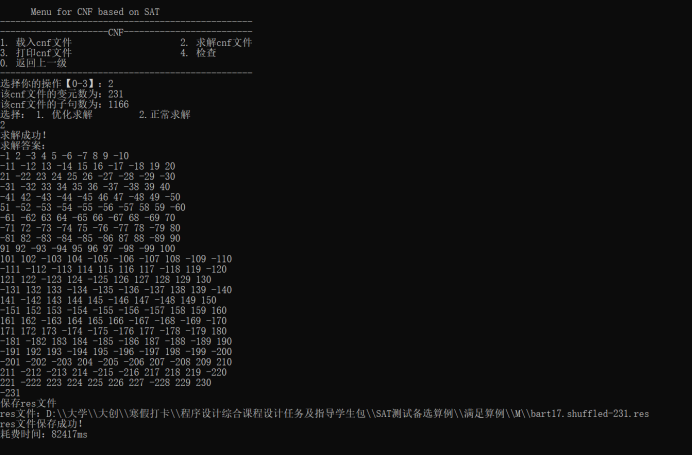


图4-15 中型算例1优化前结果

图4-16 中型算例2优化后结果



图4-17中型算例2优化前结果

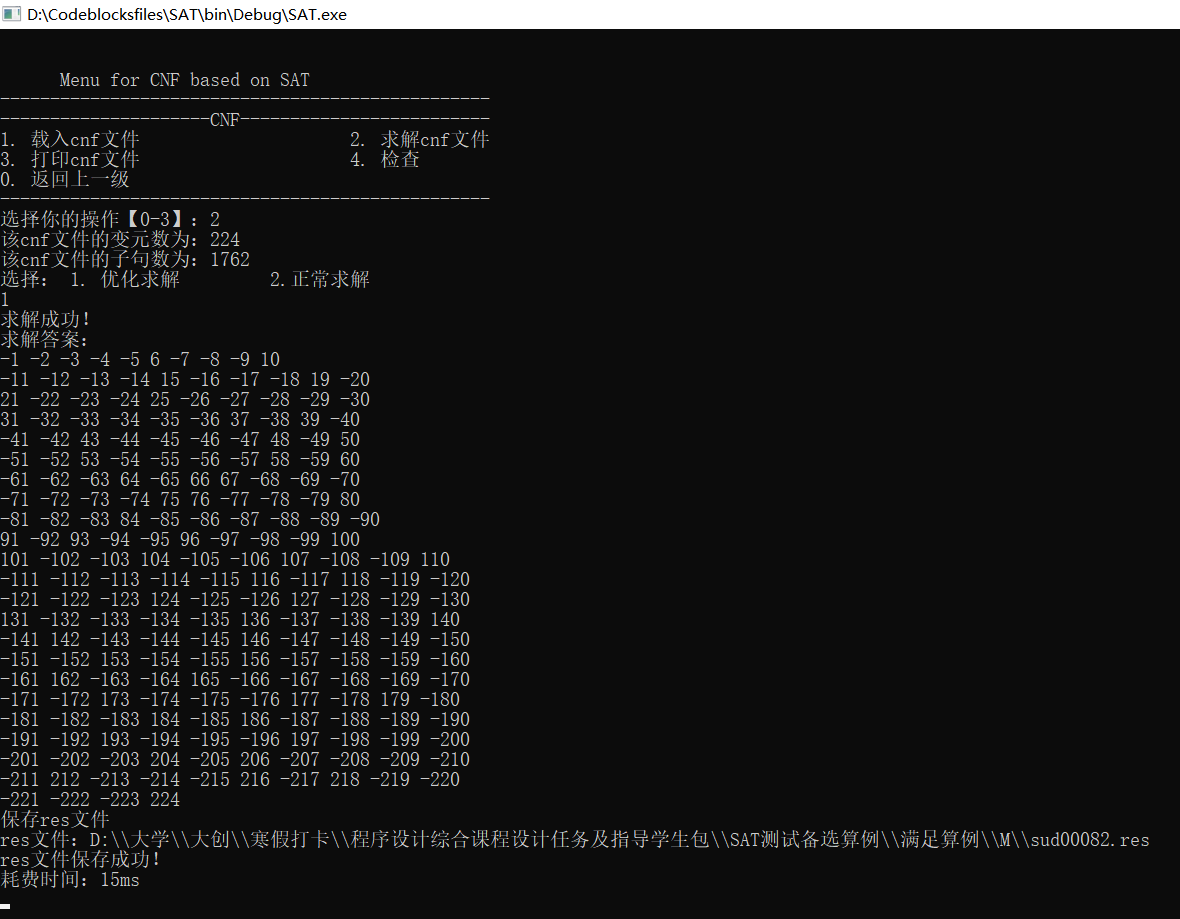


图4-18 中型算例3优化后结果

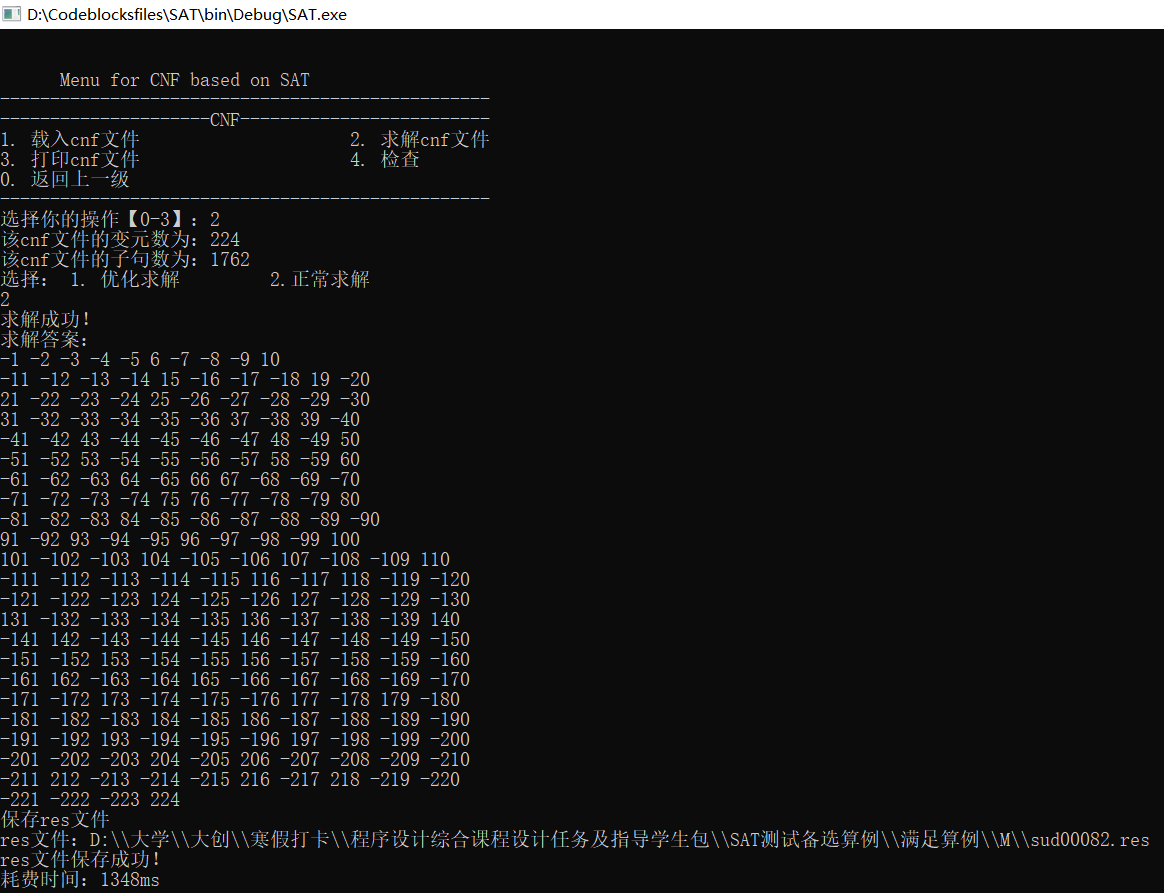


图4- 19 中型算例3优化前结果

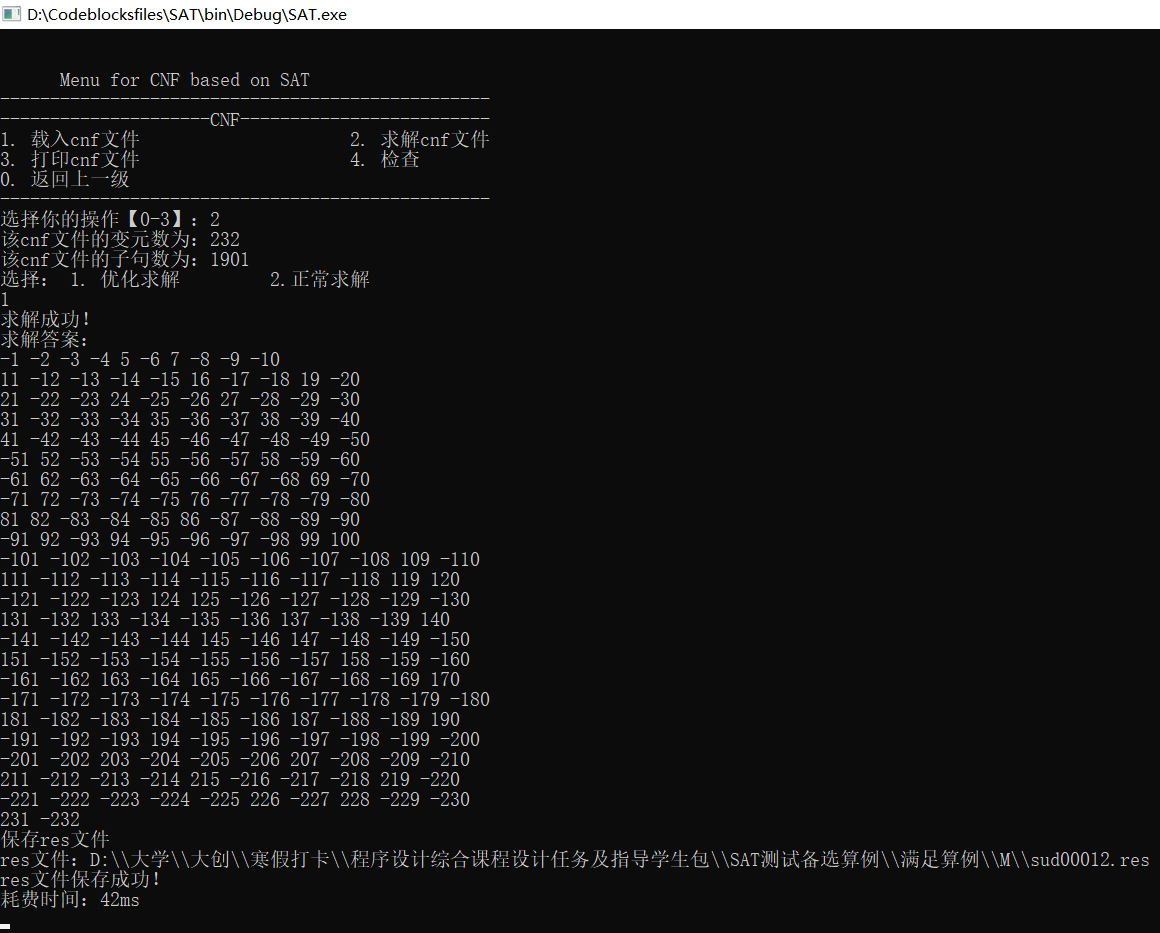


图4-20 中型算例4优化后结果

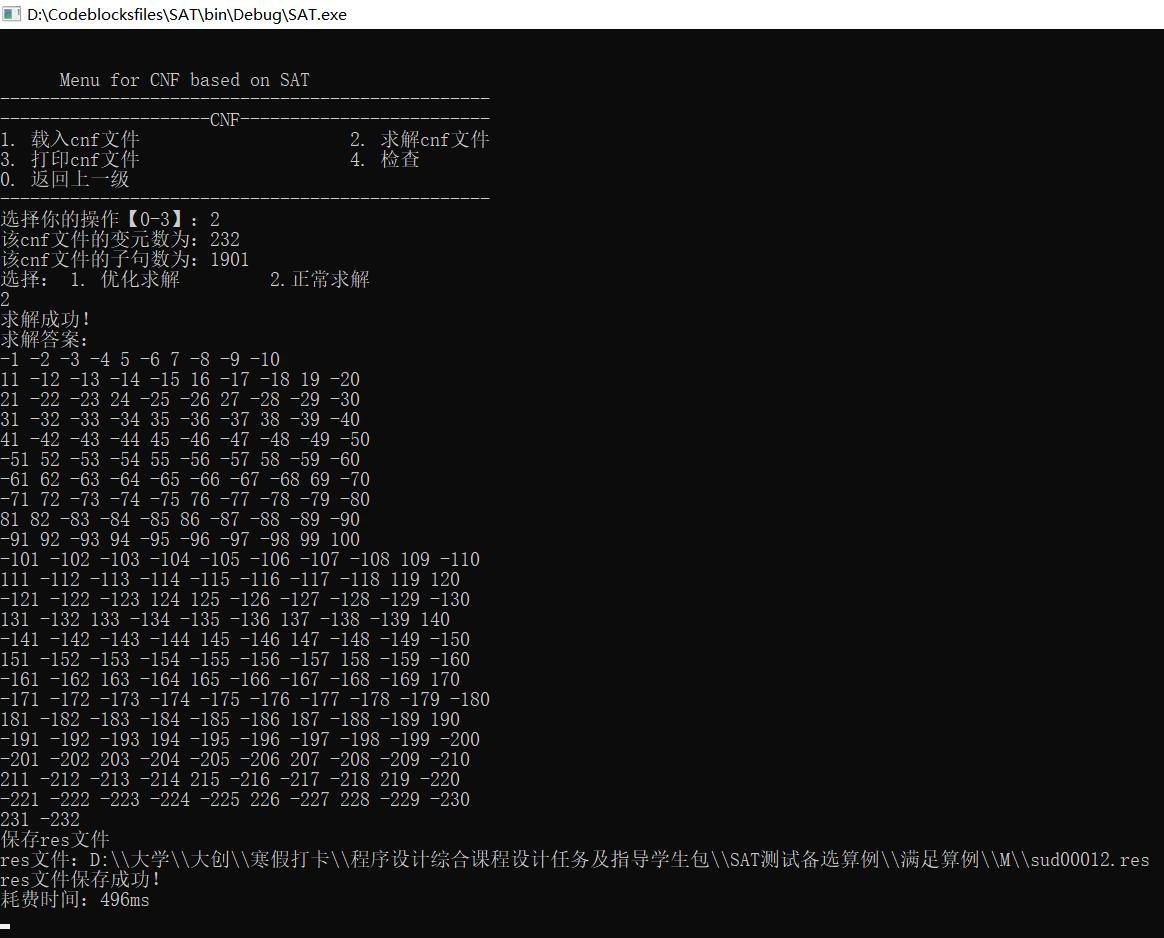


图4-21 中型算例4优化前结果

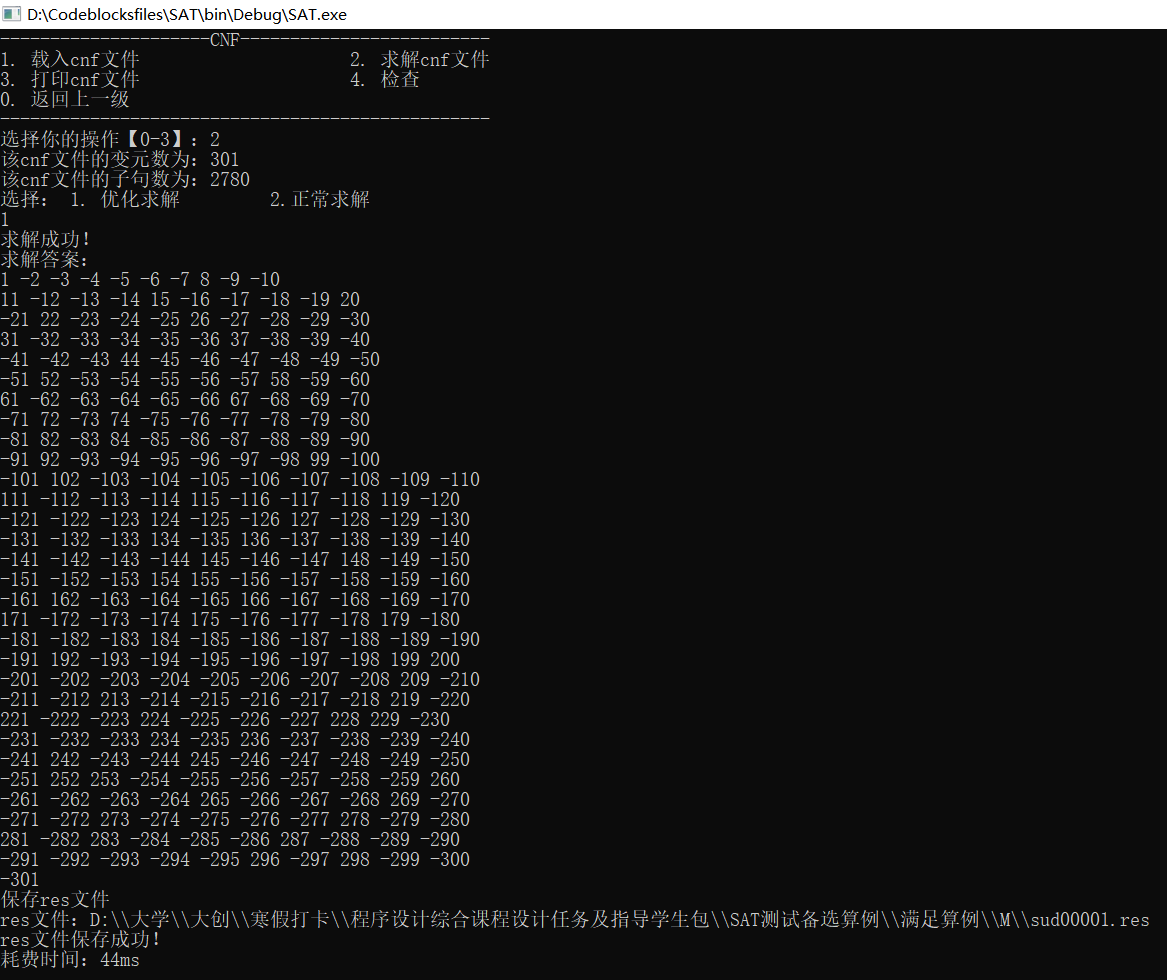


图4-22 中型算例5优化后结果

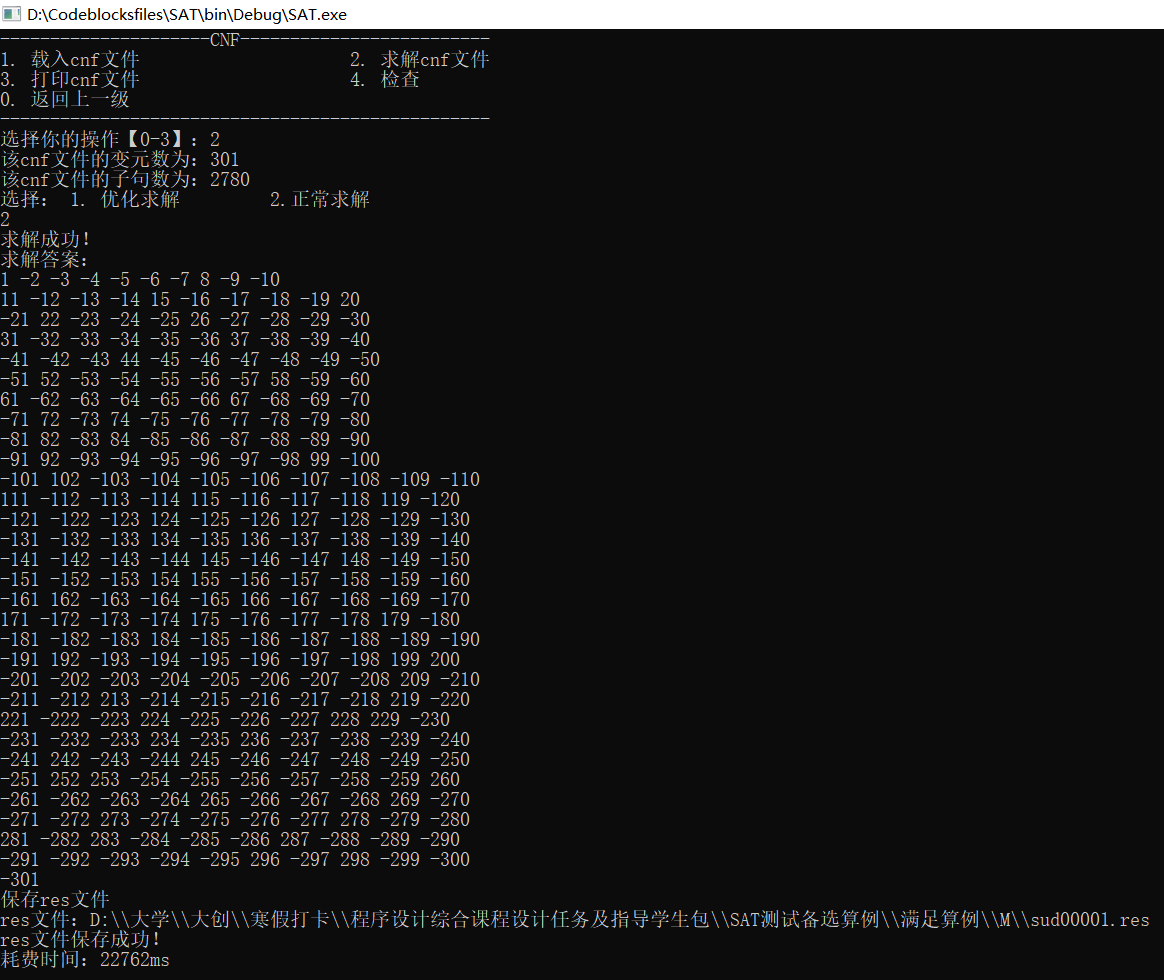


图4-23 中型算例5优化前结果

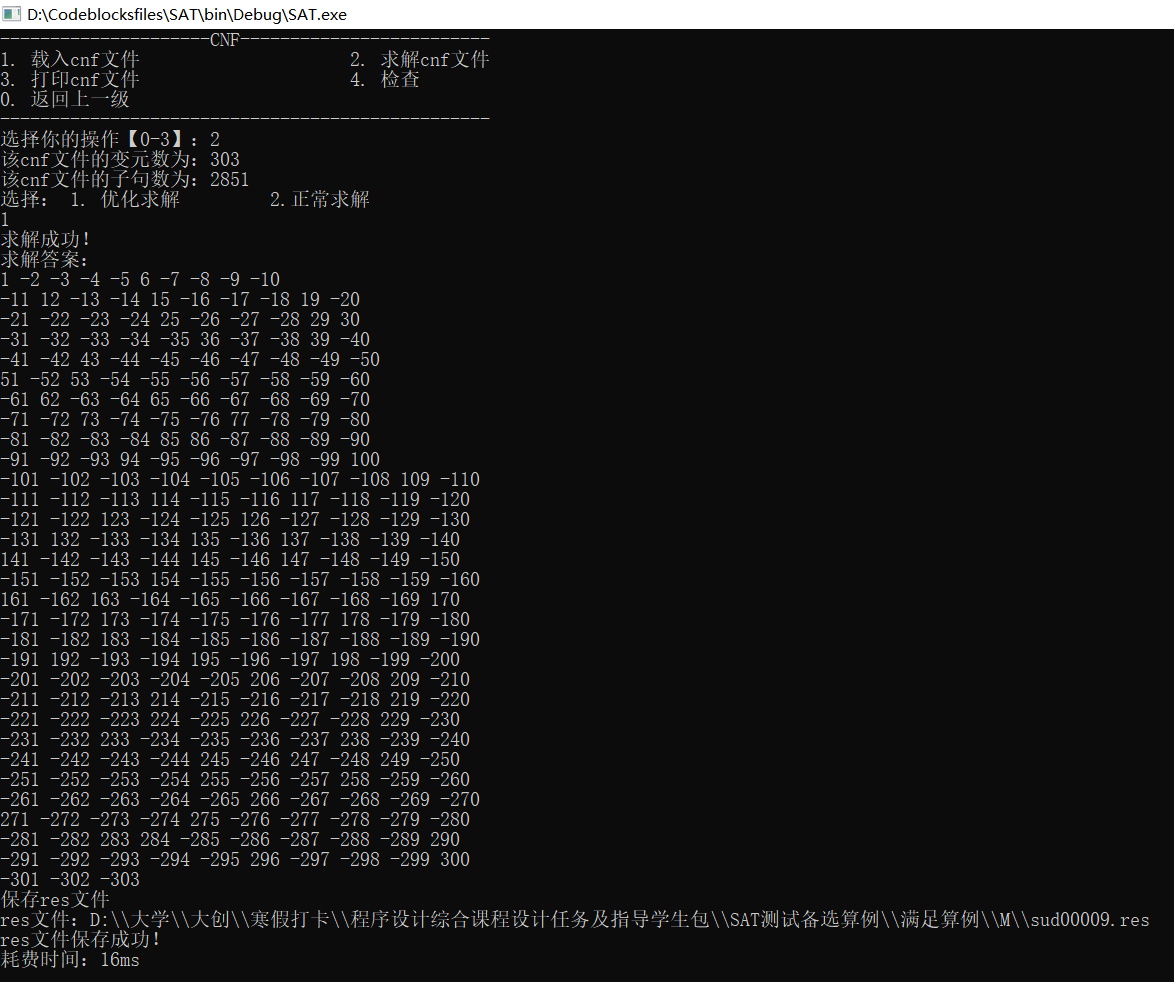


图4-24 中型算例6优化后结果

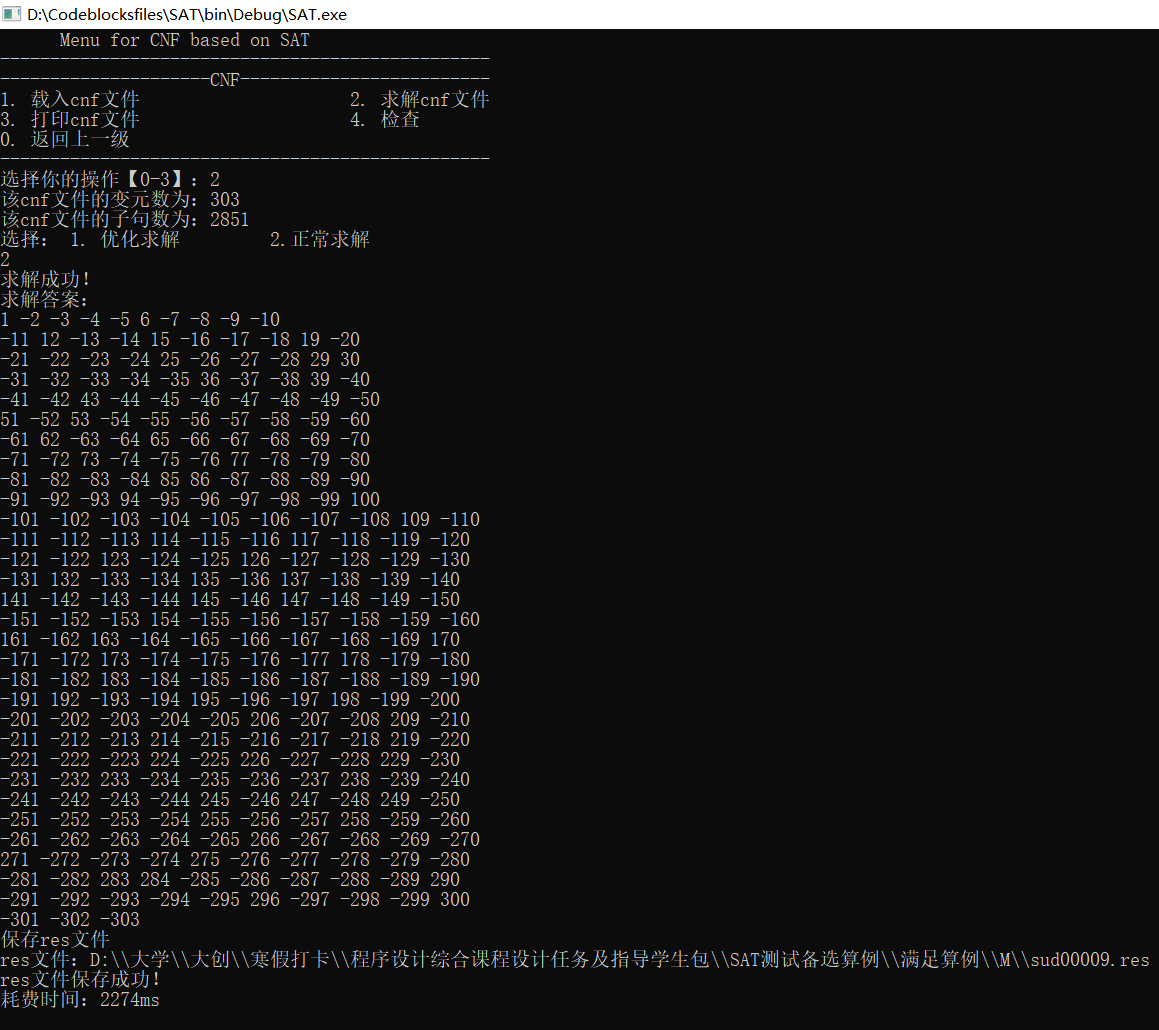


图4-25 中型算例6优化前结果

表4-3 L型算例

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变元数 | 子句数 | 子句数与变元数的比值 | 算例名称 | 可满足性 | 优化前 | 优化后 | 优化率 |
| 1584 | 16587 | 10.47 | ec-iso-ukn009.shuffled-as.sat05-3632-1584.cnf | 可满足 | 9171ms | 285ms | 96.89% |
| 1075 | 3152 | 3.27 | eh-dp04s04.shuffled-1075.cnf | 可满足 | 796ms | 69ms | 91.33% |
| 3176 | 10297 | 3.24 | par32-3.shuffled-3176.cnf | 不确定 | 240060ms | 240041ms | 不确定 |
| 625 | 76775 | 112.84 | eh-vmpc\_25.renamed-as.sat05-1913-625.cnf | 不确定 | 240048ms | 240043ms | 不确定 |
| 6498 | 130997 | 20.16 | ec-vda\_gr\_rcs\_w9.shuffled-6498.cnf | 不确定 | 240870ms | 240153ms | 不确定 |
| 1024 | 161664 | 157.874 | m-vmpc\_32.renamed-as.sat05-1919-1024.cnf | 不确定 | 240018ms | 240067ms | 不确定 |



图4-26 大型算例1优化后结果



图4-27 大型算例1 优化前结果

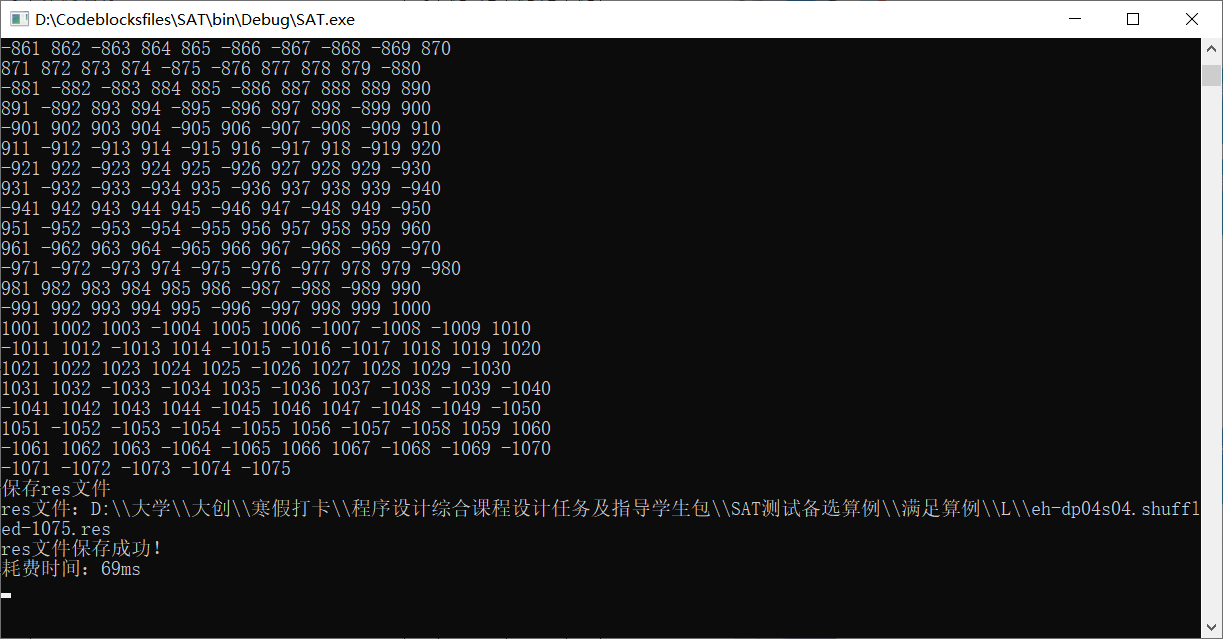


图4-28 大型算例2优化后结果

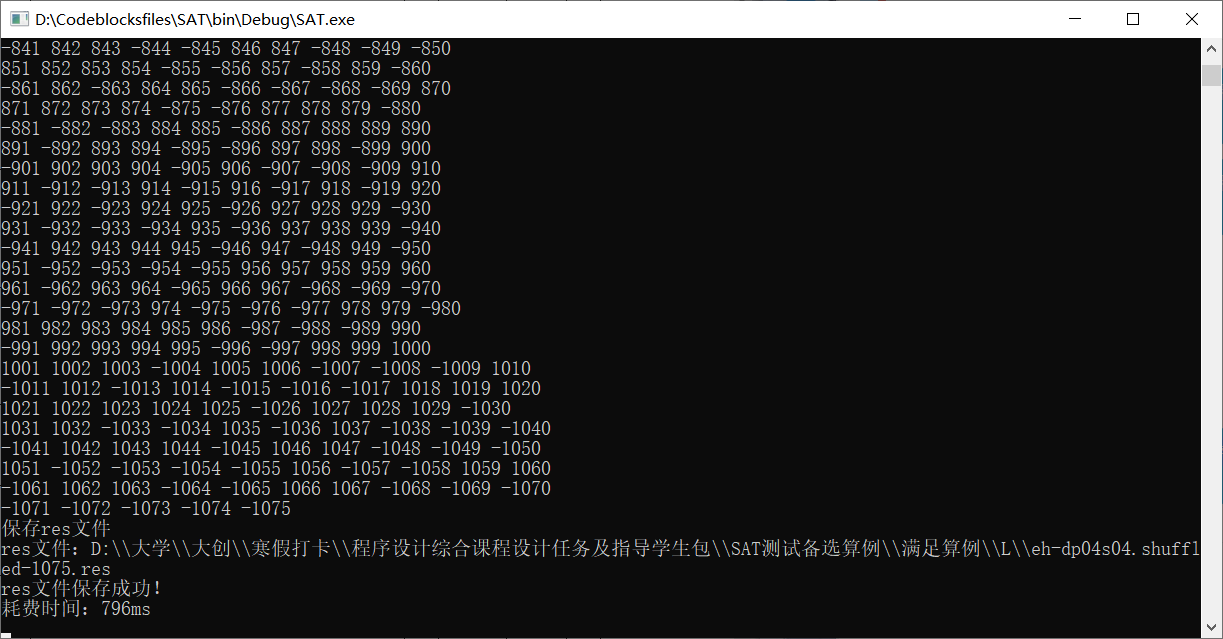


图4-29 大型算例2优化前结果



图4-30 大型算例3优化后结果



图4-31 大型算例3优化前结果



图4-32 大型算例4优化后结果



图4-33 大型算例4优化前结果

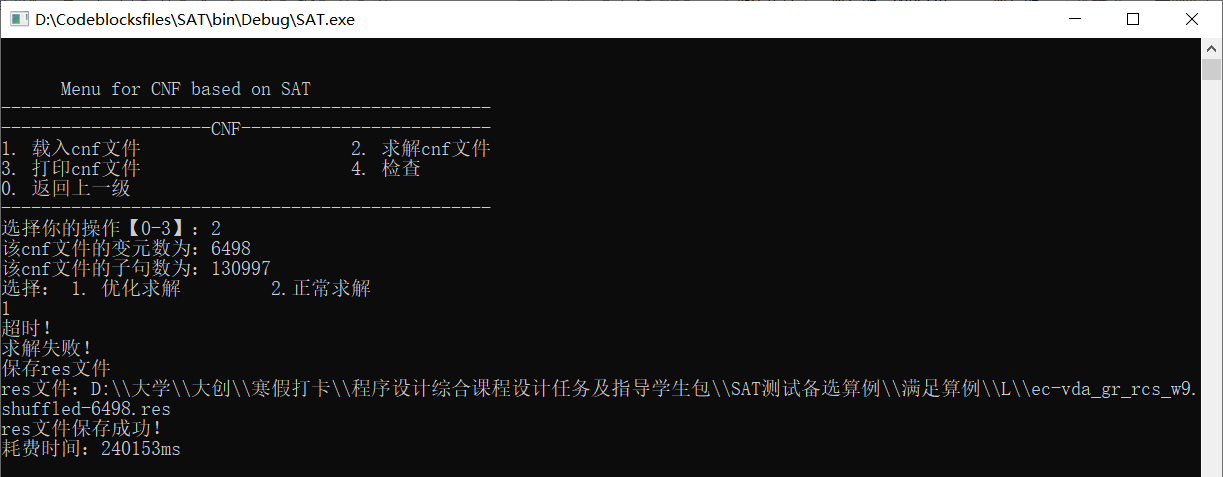


图4-34 大型算例5优化后结果



图4-35 大型算例5优化前结果



图4-36 大型算例6优化后结果

表4-4不满足算例

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 变元数 | 子句数 | 子句数与变元数的比值 | 算例名称 | 可满足性 | 优化前 | 优化后 | 优化率 |
| 10 | 100 | 10 | tst\_v10\_c100.cnf | 不可满足 | 0ms | 0ms | 不确定 |
| 50 | 100 | 2 | u-problem7-50.cnf | 不可满足 | 17ms | 0ms | 100% |
| 100 | 200 | 2 | u-problem10-100.cnf | 不可满足 | 30557ms | 0ms | 100% |

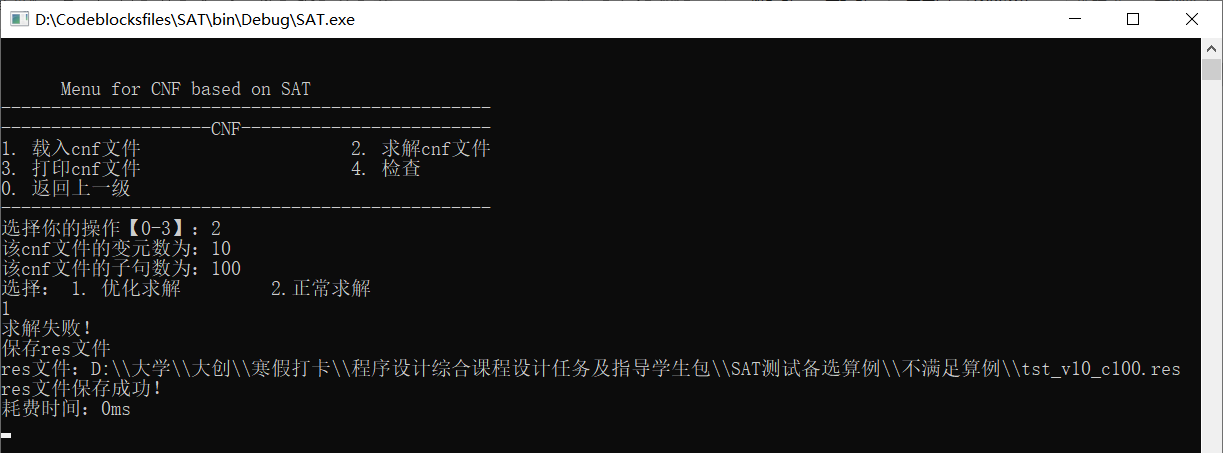


图4-37 不满足算例1优化后

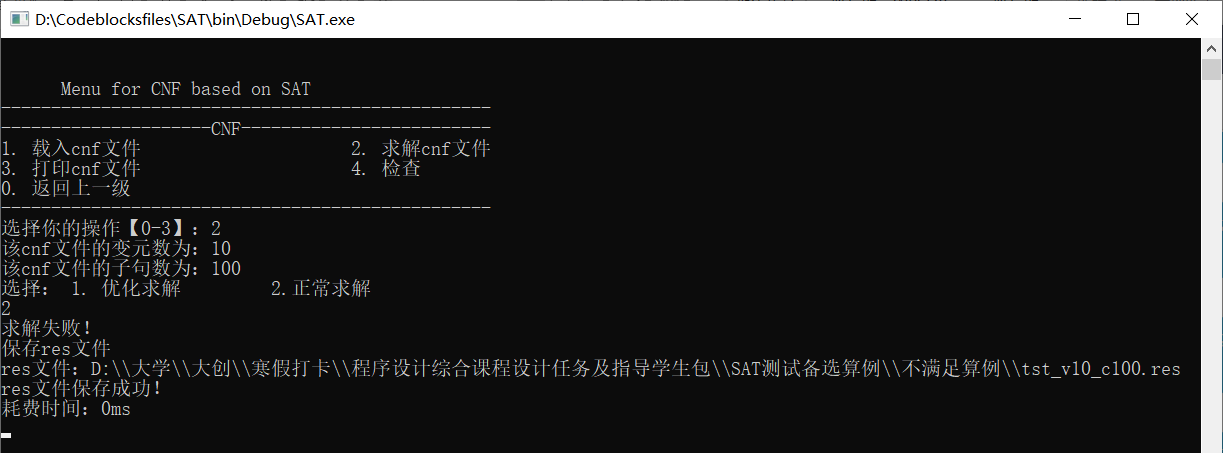


图4-38 不满足算例1优化前



图4-39 不满足算例2优化后



图4-40 不满足算例2优化前



图4-41 不满足算例3优化后

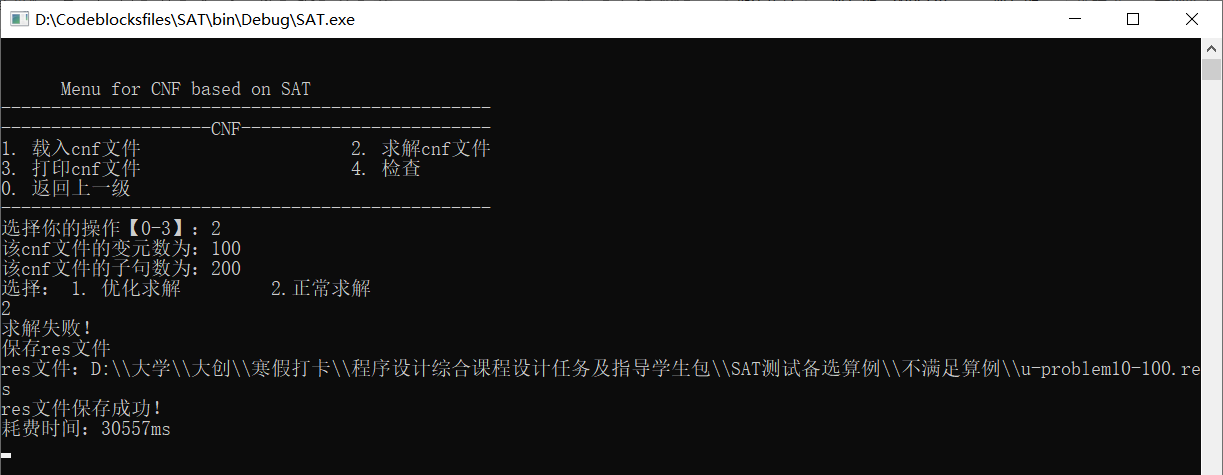


图4-42 不满足算例优化前

4.2.3 数独测试

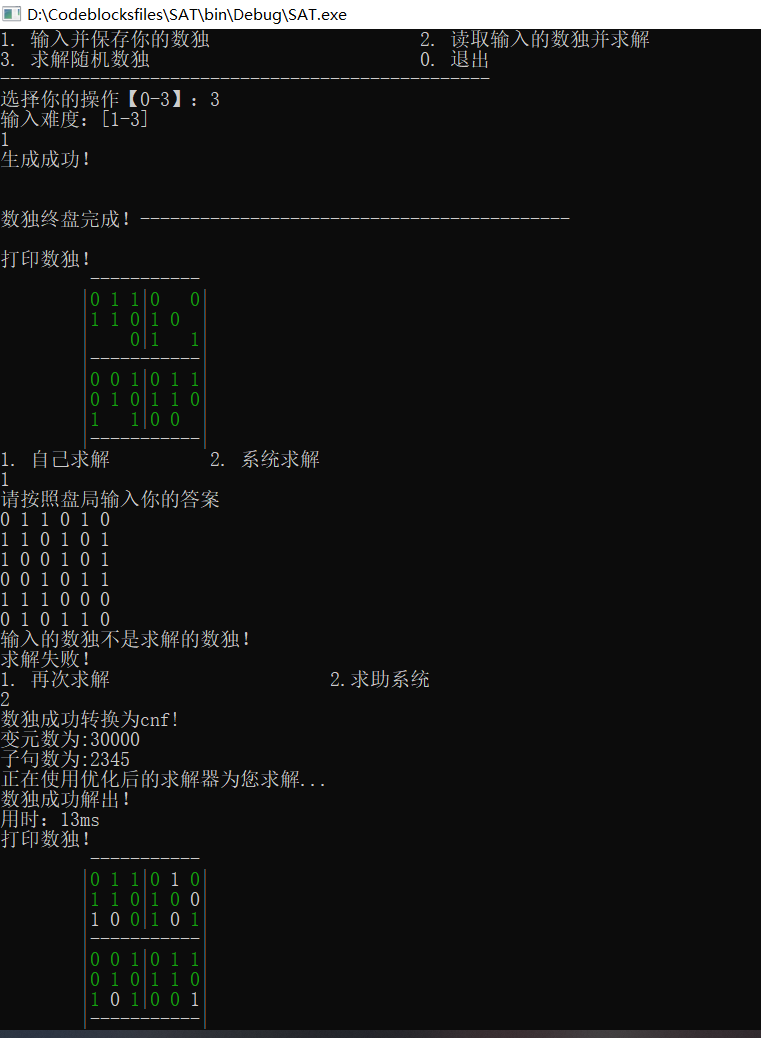


图4-43 随机数独测试结果

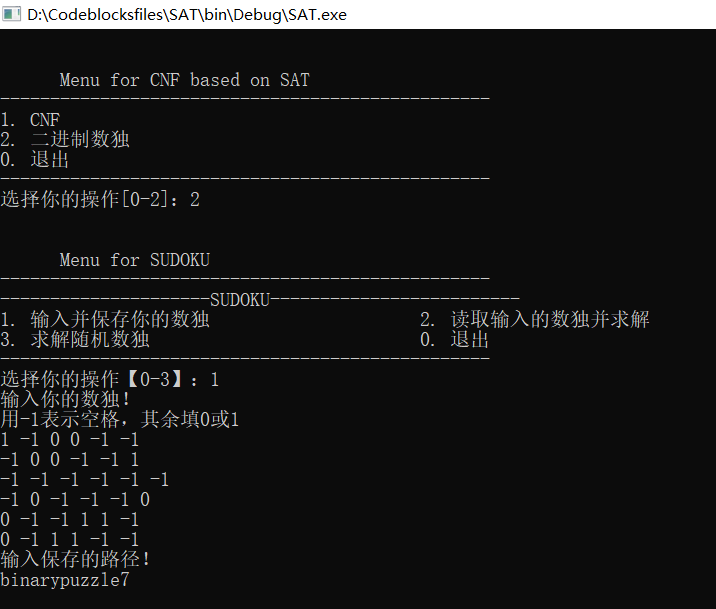


图4-44 输入自己的数独

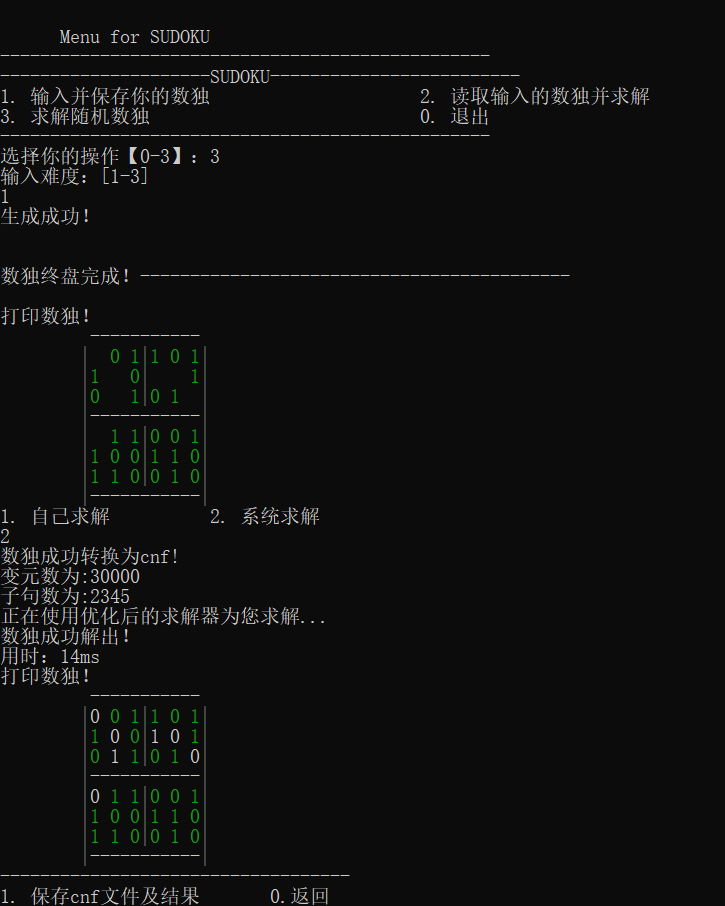


图4-45 求解输入或保存的数独

4.2.4 文件读写保存测试

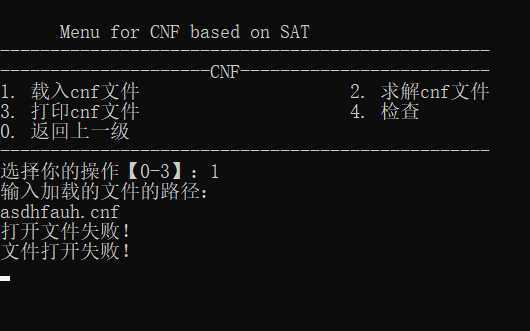


图4-46 输入不存在的cnf文件无法打开

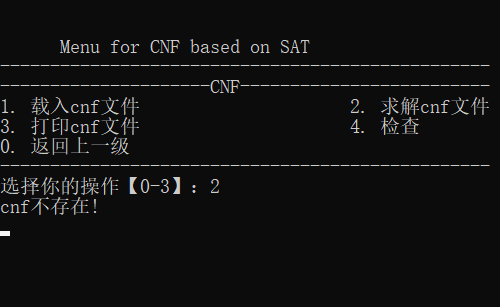


图4-47 输入错误文件后求解显示cnf不存在

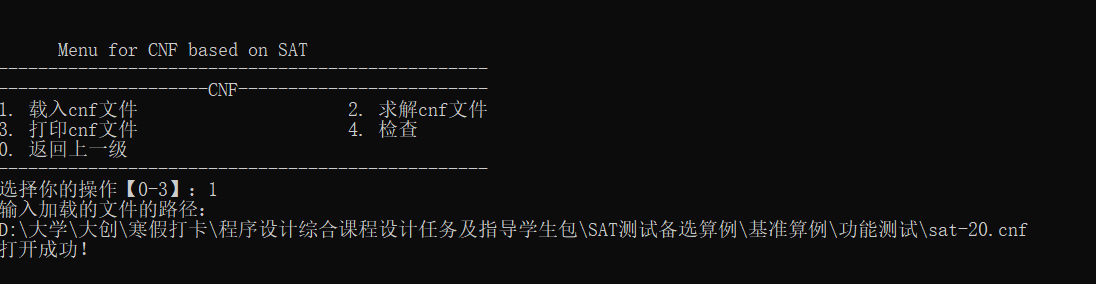


图4-48 输入正确路径的cnf文件打开成功

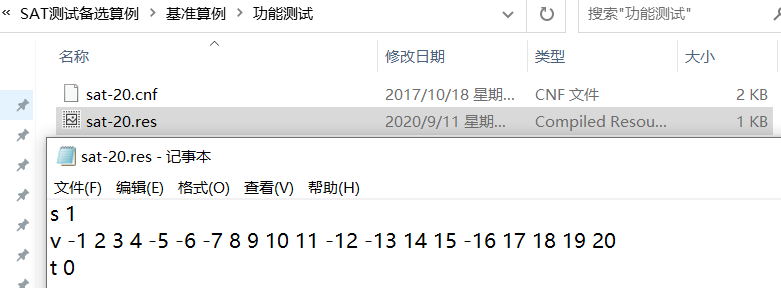


图4-49 正确求解后的的res文件

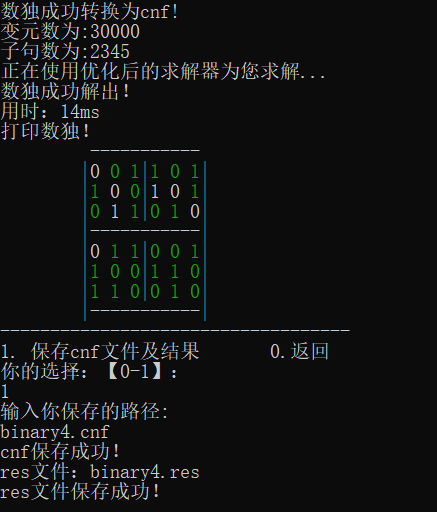




图4-50 数独转cnf文件的保存

4.3 分析

（1）SAT问题

从上面对算例运行结果可以粗略地得出：当变元数一定时，涉及到的字句数目越多，算法的优化率越低，同时，即使cnf文件的变元数目不多，但当子句数很大时（参考S型7cnf20\_90000\_90000\_7.shuffled-20.cnf变元数20，子句数1783），此时优化率明显低了很多。同时，对于算法求解时间问题，之前设置的超时时间是12000ms，结果发现许多中型算例无法确定满足性。改至24000ms后，可以看到，对子句数和变元数之比结果不大的算例是可以求解出来的且优化率较高，但是求解时间明显加长。尤其是大型算例，基本无法在短时间内算出来，通过比较可以发现子句数和变元数之比对算法的求解时间有明显的影响。此外，对同一个cnf文件测试发现，某些文件的求解时间存在较大的波动（ms量级）。

（2）数独问题

数独部分实现了随机输入6阶的二进制数独并实现求解以及随机生成一个二进制数独求解，有一定的交互性。

**5总结与展望**

5.1总结

1. 这次实验室数据结构和c语言的综合课设，实验难度明显加大。本次实验对设计者的理解能力和大局观念要求比较高，不能急着想到什么写什么，应该在进行实验之前想好做这个实验需要实现什么样的问题，实现这个问题需要用到什么样的函数，实现这些函数最好设计什么样的结构体。这次实验还有一大难点在于调试问题，这次实验的代码量非常大，而且设计的函数也很多，一旦运行结果不合期望，就会一头雾水。再加上对所使用的软件不够熟练，无法按照老师说的那样进行调试检查，检查错误无疑成了最大的问题。如果是自己检查几遍都无法检查出的错误，只好找和自己定义相同结构体的同学探讨，比较代码。这次实验其实理解算法并不难，网上也有很多论文对DPLL算法及其优化算法有详细的解释，难点在于如何实现。这次实验最大的感触就是学会怎么把一个简简单单几行的算法转化为一个可实行的程序，为了看懂算法尤其是实现算法的优化，我查阅了很多的资料，在没有进行这次实验之前，绝对没有想到需要花这么大的功夫去达到实现的目标。甚至刚开始的时候没有把程序想得全面，在编写好了子句和文字以及读写的相关函数之后，发现优化后的算法如果根据当前设置的结构体实在是无从下手，导致直接放弃重做。
2. 这次实验仍然出现了很多细节方面的问题，大体注释上有，这些问题编译不会报错但是程序运行要么异常结束，要么不合期望。在数独实现那一块，其实求解难度不大，只要转化为cnf子句正确一般就能正确求解，难点还是在于数独终盘的形成和保证他的唯一性。我出现过的主要问题是求解的结果会出现重复的行和列，这根据二进制数独的规则是不正确的，我经过几天的调试后，倒是检查出来一些小错误，但是一直没能搞懂重复的行和列问题出现在哪里（有几次是只出现了重复的列没有重复的行，所以开始怀疑是dpll算法的问题，转而去检查调试算法了）。最后还是感谢助教的帮助，仔细查看了我的代码，发现还是第三个约束转化为子句时，漏了一句。即157= ¬[1571∧1572∧…∧1578] 转化为CNF时为

（¬157∨¬1571∨¬1572∨…∨¬1578）∧（1571∨157）∧（1572∨157）…（1578∨157）形成9条子句，还要加上157这一仅有1个变元的单子句，因为当其他8个（1571，1572，...1578）都取1而157取0时，上述句子是可满足的，但是157取0又意味着第5行和第7行是相同的，所以必须加上。漏的那句在老师给的报告里面没有出现，实际上如果加上漏掉的那句，其实会把报告上中写的子句根据单子句规则直接删除掉，但却是如果深刻理解了第三个约束的内容，会发现必须加上漏掉的那句，才能保证约束三的实现。所以下次再看报告的时候，一定要理解清楚变化和实现的原理才能保证不会出现错误

（3）我是在开学前近两周就开始写的，一直到开学后第二周才勉强完成，从本次实验实现过程来看，自己之前，直到现在也是，一直拘泥与一般的函数实现，对一个完整的程序的实现缺乏系统的练习和操作。对于算法的优化看起来很复杂其实只需要改变一些细节即可得到时间上的较大优化，比如说改变变元的选择策略，将递归改为非递归，改变数据的存储方式等等。经过本次课程设计，我对算法的优化方面有了更深的理解，争取在以后的较大程序设计中采用尽可能优化的算法结构。

**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]Binary Puzzle：<http://www.binarypuzzle.com/>

[6] Putranto H. Utomo and Rusydi H. Makarim. Solving a Binary Puzzle. Mathematics in Computer Science, (2017) 11:515–526

[7] 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.

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

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

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

<http://zhangroup.aporc.org/images/files/Paper_3485.pdf>

1. Baochen SUN, Xiwei SUN, Yue WU,etc. A New Algorithm for Generating Unique-solution Sudoku. IEEE Fourth International Conference on Natural Computation,2008
2. 基于“挖洞”思想的数独游戏生成算法，薛源海, 蒋彪彬, 李永卓<https://wenku.baidu.com/view/200c448af705cc1754270957.html>
3. 基于CDCL的SAT问题求解算法研究，易念著，http://www.doc88.com/p-9089619184524.html

**附录**

main.c

#include "display.h"

int main()

{

int op=1;

srand(time(NULL));

while(op)

{

system("cls"); //清屏

printf("\n\n");

printf(" Menu for CNF based on SAT \n");

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

printf("1. CNF\n");

printf("2. 二进制数独\n");

printf("0. 退出\n");

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

printf("选择你的操作[0-2]：");

scanf("%d",&op);

switch(op)

{

case 1:

CNFpage();

getchar();getchar();

break;

case 2:

SudokuPage();

getchar();getchar();

break;

case 0:

break;

}

}

printf("欢迎再次使用!\n");

getchar();getchar();

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

clasue.h

#ifndef CLAUSE\_H\_INCLUDED

#define CLAUSE\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE -1

#define NOTSURE 0

#define NOTCONTAIN 2

#define EVALUE(x) ((x>0)?1:-1)

typedef int status;

typedef char boolean;

typedef struct Node{

int var; //正为正，负取非

struct Node\* next; //指向下一个变元

}Node; //变元定义

typedef struct Clause{

int length; //子句中变元的数目

Node\* head; //指向子句中的第一个变元

Node\* rmv; //指向第一个被移除的变元

status isremoved; //子句是否被移除子句集

struct Clause\* pre; //指向上一条子句

struct Clause\* next; //指向下一条子句

}Clause; //子句定义

typedef struct CNF{

int varnum; //变元个数

int clausenum; //子句的数目

int\* varfloor; //变元所在的决策层

int\* var; //变元取值数组，1为真，-1为假，0为不确定

int\* vsids; //VSIDS策略积分器数组

Clause\* root; //指向第一个子句

struct LearnClause\* learn\_root; //指向第一个学习子句

struct LiteralIndex\* lindex;//变元索引,指向第一个索引

}CNF; //cnf子句集定义

//当冲突产生时，导致冲突的字句会被记录下来即为学习子句，可以防止在接下来的搜索过程中再次发生同样的冲突

//利用新近学习到的冲突子句可以计算出最优的回数点

//冲突分析是先找到产生冲突的字句，然后再根据冲突的字句判断回溯到的目标层数

typedef struct LearnClause{

boolean isInStack; //判断是否在回溯栈中

int floor; //决策层数

int count; //调用次数

Clause\* clause; //指向该学习子句对应的子句

struct LearnClause\* next; //指向下一条学习子句

}LearnClause;

typedef struct LiteralIndex { //cnf索引的next指向子句索引，下标对应变元

Node\* Np; //指向与下标一致的变元

Clause\* Cp; //指向变元出现的子句

struct LiteralIndex\* next;//指向下一个索引

} LiteralIndex;

//相关函数声明

status createCNF(); //创建CNF

status destroyCNF(); //销毁CNF

status clearCNF(); //清空cnf答案

status createLIndex(); //创建变元索引

status addLIndex(); //添加索引

status deleteClauseLIndex(); //删除子句变元索引

status deleteallLIdex(); //删除所有变元索引

status addClause(); //添加子句

status insertClause(CNF\* cnf,Clause\* cl); //顶部插入子句

Clause\* removeClause(); //移除子句

status deleteClause(); //删除子句

status deleteallClause(); //删除所有子句

status deleteLearnClause(); //删除学习子句

status arrayAssign(); //变元真假取值保存到数组

boolean isClauseEmpty(); //子句是否为空

boolean isClauseUnit(); //判断是否为单元子句

boolean evaluateClause(); //子句表达式是否为真

boolean haveEmptyClause(); //是否有空子句

boolean haveClause(); //子句集是否为空

Clause\* LocateUnitClause(); //是否有单元子句并返回

status printClause(); //打印子句

status printLearnClause(); //打印学习子句

status printLIndex(); //打印索引

status addvar(); //添加变元

status deletevar(); //删除变元

status backvar(); //恢复变元

Node\* removevar(); //移除变元，并返回

status printcnf();

#endif // CLAUSE\_H\_INCLUDED

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

clause.c

#include "clause.h"

status createCNF(CNF\* cnf,int varnum,int clausenum)

{

cnf->varnum=varnum;

cnf->clausenum=clausenum;

cnf->root=NULL;

cnf->learn\_root=NULL;

cnf->var=(int\*)malloc(sizeof(int)\*(varnum+1));

cnf->vsids=(int\* )malloc(sizeof(int)\*(varnum\*2+1)); //最大分配为为每一个变元的正负都分配空间

cnf->varfloor=(int\* )malloc(sizeof(int)\*(varnum+1)); //搜索树的最大层数即为变元的数目

cnf->lindex=NULL;

if(cnf->var&&cnf->varfloor)

{

clearCNF(cnf);

return OK;

}

else

return ERROR;

}

status destroyCNF(CNF\* cnf)

{

if(!cnf)

return ERROR;

else

{

deleteallClause(cnf);

deleteallLIdex(cnf);

free(cnf->var);

free(cnf->learn\_root);

free(cnf->root);

free(cnf->varfloor);

free(cnf->vsids);

return OK;

}

}

status clearCNF(CNF\* cnf)

{

LearnClause\* lclause;

for(lclause=cnf->learn\_root;lclause;lclause=cnf->learn\_root)

{

deleteClauseLIndex(cnf,lclause->clause);

deleteLearnClause(cnf,lclause);

}

for(int i=0;i<=cnf->varnum;i++)

{

cnf->var[i]=0;

cnf->varfloor[i]=-1;

cnf->vsids[i]=cnf->vsids[i+cnf->varnum]=0;

}

return OK;

}

status addClause(CNF\* cnf,int var\_num,int\* clause)

{

//子句不存在，先要建立

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

if(!newclause||var\_num<=0)

return ERROR;

else

{

newclause->head=NULL;

newclause->rmv=NULL;

newclause->length=0; //因为是逐个变元添加的，不能再一开始就将句子的长度定义为var\_num！

newclause->isremoved=FALSE;

for(int i=0;i<var\_num;i++)

addvar(newclause,clause[i]);

if(cnf->root)

cnf->root->pre=newclause;

newclause->next=cnf->root; //不能把下面的句子放到if语句里面

newclause->pre=NULL;

cnf->root=newclause;

cnf->clausenum++;

return OK;

}

}

status createLIndex(CNF\* cnf)

{

//cnf->lindex[1].next指向变元为一的分索引

if(cnf->clausenum==0||cnf->varnum==0)

return ERROR;

if(cnf->lindex!=NULL) //删除原索引

deleteallLIdex(cnf);

else

cnf->lindex=(LiteralIndex\*)malloc(sizeof(LiteralIndex)\*(cnf->varnum+1)); //为varnum个变元建立总的索引表

Clause\* clause;

for(int i=1;i<=cnf->varnum;i++)

cnf->lindex[i].next=NULL; //下标对于变元

for(clause=cnf->root;clause;clause=clause->next)

addLIndex(cnf,clause); //为每一个变元建立分索引，每一个结构体里的Cp指向该变元出现的子句的位置

return OK;

}

//总觉得这里怪怪的

status addLIndex(CNF\* cnf,Clause\* clause)

{

if(cnf->lindex==NULL)

return FALSE; //cnf索引未建立

Node\* nodep;

LiteralIndex\* lp;

for(nodep=clause->head;nodep;nodep=nodep->next) //每次加入都是头插法

{

lp=(LiteralIndex\*)malloc(sizeof(LiteralIndex));

lp->Np=nodep;

lp->Cp=clause;

lp->next=cnf->lindex[abs(nodep->var)].next; //分索引指向总索引表上的该变元所在的后一变元

cnf->lindex[abs(nodep->var)].next=lp;

}

return OK;

}

status insertClause(CNF\* cnf,Clause\* clause)

{

//子句已存在只插入

if(cnf->root)

cnf->root->pre=clause;

clause->next=cnf->root;

clause->pre=NULL;

cnf->root=clause;

clause->isremoved=FALSE;

cnf->clausenum++;

return OK;

}

Clause\* removeClause(CNF\* cnf,Clause\* clause)

{

//只改变指针的指向但不释放空间

Clause\* pre\_clause=clause->pre;

if(clause->next) //不是最后一个

clause->next->pre=pre\_clause;

if(pre\_clause) //不是第一个

pre\_clause->next=clause->next;

else

cnf->root=clause->next;

clause->isremoved=TRUE;

cnf->clausenum--;

return clause;

}

status deleteClause(CNF\* cnf,Clause\* clause)

{

Clause\* pre\_clause=clause->pre;

if(clause->next)

clause->next->pre=pre\_clause;

if(pre\_clause)

pre\_clause->next=clause->next;

else

cnf->root=clause->next;

//释放子句的文字空间

Node\* nodep;

for(nodep=clause->head;nodep;nodep=clause->head)

{

clause->head=nodep->next;

free(nodep);

}

//记得删除移除变元内存

for(nodep=clause->rmv;nodep;nodep=clause->rmv)

{

clause->rmv=nodep->next;

free(nodep);

}

free(clause);

cnf->clausenum--;

return OK;

}

status deleteallClause(CNF\* cnf,Clause\* clause)

{

Clause\* clausep;

//释放子句空间

for(clausep=cnf->root;clausep;clausep=cnf->root)

deleteClause(cnf,clause);

//记得释放学习子句的空间

LearnClause\* lclausep;

for(lclausep=cnf->learn\_root;lclausep;lclausep=cnf->learn\_root)

{

cnf->learn\_root=lclausep->next; //没能释放变元空间

free(lclausep);

}

return OK;

}

status addvar(Clause\* clause,int var)

{

Node\* node=(Node\* )malloc(sizeof(Node));

if(!node)

return ERROR;

node->var=var;

node->next=clause->head;

clause->head=node;

clause->length++;

return OK;

}

status deleteLearnClause(CNF\* cnf,LearnClause\* lclause)

{

LearnClause\* pre\_lcp=cnf->learn\_root;

if(pre\_lcp==lclause) //第一条学习子句

cnf->learn\_root=lclause->next;

else

{

for( ;pre\_lcp&&pre\_lcp->next!=lclause;pre\_lcp=pre\_lcp->next) //找到待删的学习子句的前驱

;

}

if(!pre\_lcp)

return ERROR;

pre\_lcp->next=lclause->next;

deleteClause(cnf,lclause->clause);

free(lclause);

return OK;

}

Node\* removevar(Clause\* clause,int var)

{

//只是移除不做删除

//只改指针不该变量

Node\* nodep=clause->head;

Node\* nodeq=NULL;

// Node\* node=NULL;

//int flag=0;

for( ;nodep;nodep=nodep->next)

{

if(nodep->var==var)

{

//flag=1;

//node=nodep;

clause->length--;

if(!nodeq) //第一个移除的变元

clause->head=nodep->next; //将变元移除子句集,注意是head不是next

else

nodeq->next=nodep->next;

nodep->next=clause->rmv; //将移除的变元放入移除变元集

clause->rmv=nodep; //不能直接写clause->rmv=nodep因为无法确认之前是否有其他的已移除的子句

return nodep;

}

nodeq=nodep;

}

// if(!flag)

return NULL; //没找到该变元

// else

// return node;

}

status backvar(Clause\* clause,Node\* node)

{

//参数为指向变元的指针node及变元所在的字句clause指针

//修改移除子句

if(clause->rmv==node) //第一个变元

clause->rmv=node->next;

else

{

Node\* nodep=NULL; //前驱指针

for(nodep=clause->rmv;nodep&&nodep->next!=node;nodep=nodep->next)

;

if(!nodep)

return ERROR;

nodep->next=node->next;

}

//将恢复的变元放插入子句集

node->next=clause->head;

clause->head=node;

clause->length++;

return OK;

}

status deletevar(Clause\* clause,int var)

{

Node\* nodep=NULL;

Node\* nodeq=NULL;

for(nodep=clause->head;nodep;nodep=nodep->next)

{

if(nodep->var==var)

{

clause->length--;

if(!nodeq) //第一个

clause->head=nodep->next;

else

nodeq->next=nodep->next;

free(nodep);

return OK;

}

nodeq=nodep;

}

return ERROR;

}

status deleteClauseLIndex(CNF\* cnf,Clause\* clause)

{

LiteralIndex\* lindex;

Node\* node;

LiteralIndex\* temp;

//修改子句集中的索引

for(node=clause->head;node;node=node->next)

{

//找到变元所在的索引位置的前驱

for(lindex=&(cnf->lindex[abs(node->var)]);lindex->next->Cp!=clause;lindex=lindex->next)

;

temp=lindex->next;

lindex->next=temp->next; //不能直接写lindex->next=lindex->next->next！！！

free(temp);

}

//不在子句集中，修改移除集中的索引

for(node=clause->rmv;node;node=node->next)

{

for(lindex=&(cnf->lindex[abs(node->var)]);lindex->next->Cp!=clause;lindex=lindex->next)

;

temp=lindex->next;

lindex->next=temp->next;

free(temp);

}

return OK;

}

status deleteallLIdex(CNF\* cnf,Clause\* clause)

{

if(cnf->lindex==NULL)

return OK;

for(int i=1;i<=cnf->varnum;i++)

{

while(cnf->lindex[i].next)

{

//搞不懂啥意思

LiteralIndex\* lindex=cnf->lindex[i].next;

cnf->lindex[i].next=lindex->next;

free(lindex);

}

}

return OK;

}

status arrayAssign(CNF\* cnf,int var,int floor)

{

cnf->var[abs(var)]=EVALUE(var);

cnf->varfloor[abs(var)]=floor;

return OK;

}

status printClause(CNF\* cnf)

{

printf("布尔变元数目:%d\n",cnf->varnum);

printf("子句数目：%d\n",cnf->clausenum);

printf("各变元取值:\n");

for(int i=1;i<=cnf->varnum;i++)

{

printf("第%d个：%d ",i,cnf->var[i]);

if(i%10==0)

printf("\n");

}

printf("决策层数：\n");

for(int i=1;i<=cnf->varnum;i++)

printf("第%d个变元位于第%d层\n",i,cnf->varfloor[i]);

return OK;

}

boolean haveEmptyClause(CNF\* cnf)

{

//判断是否含有空子句

Clause\* clause;

for(clause=cnf->root;clause;clause=clause->next)

if(clause->length==0)

return TRUE;

return FALSE;

}

boolean haveClause(CNF\* cnf)

{

//判断是否为空

return (cnf->clausenum)?TRUE:FALSE;

}

boolean evaluateClause(Clause\* clause,int var)

{

//选择一个变元var，并赋值为真，判断子句是否满足

Node\* node;

int flag=NOTCONTAIN;

for(node=clause->head;node;node=node->next)

{

if(node->var==var) //存在取值为真的变元，析取范式（子句）表达式为真

return TRUE;

else if(node->var==-var)

flag=NOTSURE; //存在该真变元的非变元，无法确定表达式的真假

}

return flag;

}

boolean isClauseUnit(Clause\* clause)

{

return (clause->length==1)?TRUE:FALSE;

}

Clause\* LocateUnitClause(CNF\* cnf)

{

Clause\* clause;

for(clause=cnf->root;clause;clause=clause->next)

if(clause->length==1)

return clause;

return NULL;

}

boolean isClauseEmpty(Clause\* clause)

{

return (clause->length)?FALSE:TRUE;

}

status printcnf(CNF\* cnf)

{

printf("\n");

for(Clause\* clause=cnf->root;clause;clause=clause->next)

{

for(Node\* node=clause->head;node;node=node->next)

printf("%d ",node->var);

printf("0\n");

}

printf("\n");

return OK;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dpll.h

#ifndef DPLL\_H\_INCLUDED

#define DPLL\_H\_INCLUDED

#include "clause.h"

#include <math.h>

#define CLAUSE 0

#define LITERAL 1

#define SPLIT 2

typedef struct Stack{

int tag; //标识子句或变元

int floor; //决策层数

Node\* Np;

Clause\* Cp;

struct Stack\* next; //下一个修改区

}ChangeStack; //保存修改

typedef struct Floor{

int a; //冲突变元

int floor; //变元所在的决策层

}Floor;

status DPLL();

status advancedDPLL();

status SimplifySingleClause(); //化简单子句

status MOM(CNF\* cnf);

status VSIDSS(CNF\* cnf);

status combineStrategy();

status saveChange(); //保存修改

status backChange(); //恢复修改

status createLearnClause(CNF\* cnf,int\* a,int i,int x); //生成学习子句

status backLearnClause(CNF\* cnf,int floor); //学习子句回溯

status deleteRepeateLearnClause(); //删除重复的学习子句

status backAssign(CNF\* cnf,int floor); //赋值回溯

status compare\_Des(const void\* a,const void\* b);

status check();

#endif //DPLL\_H\_INCLUDED

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dpll.c

#include "DPLL.h"

#include <time.h>

#define DEBUG2 0

#define VSIDS 2

#define VSIDSCOUNT 100

#define LEARNLENGTH\_MAX 20 //限制学习子句的数目

#define OUTTIME 240000

int beginTime;

status reset=FALSE; //重启，即清除现在所有变量的赋值状态，重新选择一组决策变量进行赋值然后进行正常的搜索过程

int MAXC = 10;

enum strategy {mom, vsids};

status saveChange(ChangeStack\* head,int tag,Clause\* clause,Node\* node)

{

//head为头指针

if(!head)

return ERROR;

ChangeStack\* newchange=(ChangeStack\*)malloc(sizeof(ChangeStack));

if(!newchange)

return ERROR;

newchange->tag=tag;

switch(tag)

{

case CLAUSE:

newchange->Cp=clause;

newchange->Np=NULL;

break;

case LITERAL: //变元改变意味着子句也发生改变

newchange->Np=node;

newchange->Cp=clause;

break;

case SPLIT: //作为单子句

newchange->Cp=clause;

break;

default:

break;

}

newchange->next=head->next;

head->next=newchange;

return OK;

}

status SimplifySingleClause(CNF\* cnf,int var,ChangeStack\* head)

{

Node\* node;

Clause\* clausep;

Clause\* clause=cnf->root;

while(clause)

{

//如果该字句是可满足的，则删除该字句，若无法确定，则删除这个文字

switch(evaluateClause(clause,var))

{

case TRUE:

removeClause(cnf,clause);

if(head) //保存修改！

saveChange(head,CLAUSE,clause,NULL);

clause=clause->next;

break;

case NOTSURE:

node=removevar(clause,-var); //-var!!!

if(head)

saveChange(head,LITERAL,clause,node);

//若移除变元后子句变为单子句或者空子句，为了操作方便，将其置于顶部，同时返回空子句

if(clause->length<=1)

{

clausep=clause->next;

removeClause(cnf,clause);

insertClause(cnf,clause);

if(clause->length==0)

return ERROR; //化简之后存在空子句，返回失败

// else

clause=clausep;

}

else

clause=clause->next;

break;

default:

clause=clause->next;

break;

}

}

return OK;

}

status advancedSimplifySingleClause(CNF\* cnf,Clause\* clause,ChangeStack\* head)

{

int var=clause->head->var;

int index=abs(var);

LiteralIndex\* lindex; //指向当前变元所在的索引

Clause\* clausep;

Node\* node;

removeClause(cnf,clause); //移除插入的单子句

saveChange(head,CLAUSE,clause,NULL);

for(lindex=cnf->lindex[index].next;lindex;lindex=lindex->next) //通过索引找到含有变元var的子句

{

clausep=lindex->Cp; //clausep初始化为指向含有变元var的第一个子句（创建索引时的第一个）

node=lindex->Np;

if(clausep->isremoved==FALSE)

{

if(node->var==var)

{

removeClause(cnf,clausep); //删除包含L的子句，

if(head)

saveChange(head,CLAUSE,clausep,NULL); //保存修改

}

else if(node->var==-var) //删除子句中含有非L的文字

{

node=removevar(clausep,-var);

if(head)

saveChange(head,LITERAL,clausep,node);

//将单子句和空子句置顶，节省搜索时间

if(clausep->length<=1)

{

removeClause(cnf,clausep);

insertClause(cnf,clausep);

if(clausep->length==0)

return ERROR;

}//end of if

}//end of else if

}//end of if

}//end of for

return OK;

}

status backAssign(CNF\* cnf,int floor)

{

for(int i=1;i<=cnf->varnum;i++)

{

if(cnf->varfloor[i]==floor) //找到该层的变元

{

//回复数据

cnf->var[i]=NOTSURE;

cnf->varfloor[i]=-1;

}

}

return OK;

}

status MOM(CNF\* cnf)

{

//最短子句出现频率最大优先法

//子句长度越大越容易满足，因此从较短子句中挑选出较大频率的文字使其优先满足

//算法的重点是求解出和文字l相对应的J指，最终选J值最大的文字l赋值为真

//ni记为包含文字l的字句Ci的字句的长度，则J值即为2的负ni次方求和,长度越长,J值越小

double\* J=(double\*)malloc(sizeof(double)\*(cnf->varnum+1));

for(int i=1;i<=cnf->varnum;i++) //初始化为0

J[i]=0;

Clause\* clause;

Node\* node;

double max=-1;

int index;

//printcnf(cnf);

for(clause=cnf->root;clause;clause=clause->next)

for(node=clause->head;node;node=node->next)

J[abs(node->var)] +=1.0/(1<<(clause->length));

for(int i=1;i<=cnf->varnum;i++)

{

if(max<J[i]&&cnf->var[i]==NOTSURE)

//寻找最大的J值对应的变元(只确定变元的绝对值，不确定是真或者非）

{

max=J[i];

index=i;

}

}

//确定取真或者取非,看子句集中哪一种取得多,频率最大

// printf("index为：%d\n",index);

int positive=0;

int negative=0;

for(clause=cnf->root;clause;clause=clause->next)

for(node=clause->head;node;node=node->next)

{

if(node->var>0)

positive++;

else

negative++;

}

free(J);

return (positive>negative)?index:(-index);

}

status VSIDSS(CNF\* cnf)

{

// 为每一个变元的正负文字各设一个计数器，初始值设为在子句集种的出现次数

//根据未赋值变量中的文字计数器的值的排序，从中选择一个是计数器值最大的一个文字作为决策变量

int max=0;

int index;

static int count=0; //记录决策次数

for(int i=1;i<=cnf->varnum;i++)

if(max<=cnf->vsids[i]+cnf->vsids[i+cnf->varnum]) //计数该文字出现次数（正和非）

if(cnf->var[i]==NOTSURE) //找到最大

{

max=cnf->vsids[i]+cnf->vsids[i+cnf->varnum];

index=i;

}

count++;

if(count>=VSIDSCOUNT)

{

//为了防止一些变量长时间得不到赋值，经过一定的决策次数之后，每一个文字的计数器的值都会除以一个常数

count=0;

for(int i=1;i<=cnf->varnum\*2;i++)

cnf->vsids[i] /=2;

}

return (cnf->vsids[index]>cnf->vsids[index+cnf->varnum])?index:(-index);

}

status combineStrategy(CNF\* cnf,enum strategy s)

{

int i=0;

switch(s)

{

case mom:

i=MOM(cnf);

break;

case vsids:

i=VSIDSS(cnf);

break;

}

return i;

}

status DPLL(CNF\* cnf,int f)

{

//f为决策层，采用递归的思想，从f=0开始往下走

Clause\* clause;

ChangeStack st;

st.next=NULL;

int v;

if(f==0) //第一层，开始算法

beginTime=clock();

while((clause=LocateUnitClause(cnf))!=NULL) //存在单子句

{

arrayAssign(cnf,clause->head->var,f);

SimplifySingleClause(cnf,clause->head->var,&st); //利用单子句规则化简子句集S

if(haveEmptyClause(cnf)==TRUE) //如果S存在空子句

{

backChange(cnf,&st,-1,NULL);

backAssign(cnf,f);

if(clock()-beginTime>=OUTTIME)

{

printf("超时！\n");

return NOTSURE;

}

return FALSE;

}

else if(haveClause(cnf)==FALSE) //S为空，满足

{

backChange(cnf,&st,-1,NULL);

return TRUE;

}

}//end of while

//选择策略选择变元v

v=combineStrategy(cnf,mom);

arrayAssign(cnf,v,f);

addClause(cnf,1,&v); //变元v单独作为子句加入到子句集S中,第二个参数为变元数目不是变元的值！！！

saveChange(&st,SPLIT,cnf->root,NULL);

int ans=DPLL(cnf,f+1);

if(ans==TRUE)

{

backChange(cnf,&st,-1,NULL);

return TRUE;

}

if(ans==NOTSURE)

{

backChange(cnf,&st,-1,NULL);

backAssign(cnf,f);

return NOTSURE;

}

backChange(cnf,&st,1,NULL);

v=-v; //取非

arrayAssign(cnf,v,f);

addClause(cnf,1,&v); //变元v单独作为子句加入到cnf中

saveChange(&st,SPLIT,cnf->root,NULL);

ans=DPLL(cnf,f+1);

backChange(cnf,&st,-1,NULL);

if(ans==FALSE)

backAssign(cnf,f);

return ans;

}

status advancedDPLL(CNF\* cnf,int f)

{

//优化后的dpll算法，即cdll算法

int back=f;//回溯的层数

Clause\* clause;

ChangeStack st;

st.floor=f;

st.next=NULL;

int ans;

int count=0;

//VSIDS策略

if(f==0)

{

beginTime=clock();

Node\* node;

for(clause=cnf->root;clause;clause=clause->next)

for(node=clause->head;node;node=node->next)

{

if(node->var>0)

cnf->vsids[node->var]++; //统计正文字

else

cnf->vsids[cnf->varnum-node->var]++; //负文字

}

}

//在求解是，由于最初的变量决策顺序可能不是最优的，这就会导致在搜索陷入某些子句空间而白白消耗时间

//此时采用重新启动机制，即清楚目前所有变量的赋值状态，重新选择一组决策变量进行赋值

do

{

count++;

if(reset==TRUE)

{

//重启动

reset=FALSE;

LearnClause\* lclause=cnf->learn\_root;

LearnClause\* lclausep=NULL;

while(lclause)

{

//子句删除机制，删除一些很长的学习子句

if(lclause->clause->rmv==NULL&&lclause->clause->length>LEARNLENGTH\_MAX)

{

//删除那些没有移除变元也就是没有赋值过的变元，并且长度很长的学习子句

lclausep=lclause->next;

deleteClauseLIndex(cnf,lclause->clause);

deleteLearnClause(cnf,lclause);

lclause=lclausep;

}

else

lclause=lclause->next;

}

}

while((clause=LocateUnitClause(cnf))!=NULL) //单子句规则简化

{

arrayAssign(cnf,clause->head->var,f);

advancedSimplifySingleClause(cnf,clause,&st);

if(haveEmptyClause(cnf)==TRUE)

{

if(clock()-beginTime>=OUTTIME)

{

printf("超时！\n");

return -1-f;

}

int\* a=(int\* )malloc(sizeof(int)\*(cnf->varnum+3)); //包含总数，变元，唯一蕴含点

backChange(cnf,&st,-1,a); //还原，第一次回复应该是删除后的子句变空，产生冲突

//若发生冲突，添加学习子句并进行非时序回溯

back=a[0];

free(a);

backLearnClause(cnf,f);

//赋值回溯

backAssign(cnf,f);

if(count>MAXC)

{

MAXC++;

//重启动

reset=TRUE;

//学习子句回溯

return -f;

}

return back-f;

}

else if(haveClause(cnf)==FALSE)

{

backChange(cnf,&st,-1,NULL);

backLearnClause(cnf,f);

return TRUE;

}

}//end of while

int v=combineStrategy(cnf,mom);

addClause(cnf,1,&v);

saveChange(&st,SPLIT,cnf->root,NULL);

ans=advancedDPLL(cnf,f+1);

backChange(cnf,&st,1,NULL);

}while(ans==FALSE);

if(ans==TRUE)

{

backChange(cnf,&st,-1,NULL);

backLearnClause(cnf,f);

return TRUE;

}

//由返回值决定回溯层数

else if(ans!=0)

{

backChange(cnf,&st,-1,NULL);

backLearnClause(cnf,f);

backAssign(cnf,f);

return ans+1;

}

else if(ans==0)

return ans;

}

status createLearnClause(CNF\* cnf,int\* a,int i,int x)

{

//采用非时序回溯，当冲突发生，不是简单地回到父节点

//采用蕴含图，找到与冲突所关联的决策变量赋值的层数，选出最大的即为所要返回的层数

//传入的参数分别为：cnf文件，冲突变元数组，冲突变元的个数，变元x

int j,s;

int\* copy=a;

int floor=0;

int backtofloor=0;

for(j=i-1;j>=0;j--)

if(floor<cnf->varfloor[abs(copy[j])])

floor=cnf->varfloor[abs(copy[j])]; //降低决策层数，即减少赋值的次数

for(s=0,j=0;j<i;j++)

{

if(cnf->varfloor[abs(copy[j])]!=floor) //删除本层已经赋值的变元

{

copy[s]=copy[j];

s++;

}

}

for(j=0;j<s;j++) //此时cpoy数组中最大的变元决策层数即为要返回的决策层

if(backtofloor<cnf->varfloor[abs(copy[j])])

backtofloor=cnf->varfloor[abs(copy[j])];

//学习子句太长，化为单子句，在回溯时删除

int pre\_length=s+1; //学习子句原长

Floor\* F=(Floor\*)malloc(sizeof(Floor)\*s);

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

{

F[i].a=copy[i];

F[i].floor=cnf->varfloor[abs(copy[i])];

}

qsort(F,s,sizeof(Floor),compare\_Des); //将变元的决策层变为降序

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

copy[i]=F[i].a;

free(F);

copy[s]=-x; //加入当前层的决策变量x（x取真时发生冲突，所以学习子句中应该取非在加入）

s++;

int temp;

for(j=s-1;j>0;j--)

{

temp=copy[j-1];

copy[j-1]=copy[j];

copy[j]=temp;

}

addClause(cnf,s,copy); //将学习子句插入到cnf文件中

LearnClause\* lclause=(LearnClause\* )malloc(sizeof(LearnClause));

lclause->clause=cnf->root;

lclause->count=pre\_length;

lclause->floor=floor;

lclause->isInStack=FALSE;

lclause->next=cnf->learn\_root;

cnf->learn\_root=lclause;

//vsids，加入学习子句后，vsids数组同时更新

for(i=0;i<pre\_length;i++)

{

if(copy[i]>0)

cnf->vsids[copy[i]]++;

else

cnf->vsids[cnf->varnum-copy[i]]++;

}

//添加学习子句的索引

addLIndex(cnf,lclause->clause);

//学习子句变元有赋值

while(cnf->root->head)

removevar(cnf->root,cnf->root->head->var);

return backtofloor;

}

status backLearnClause(CNF\* cnf,int floor)

{

Node\* node;

LearnClause\* lclause=cnf->learn\_root;

while(lclause)

{

if(lclause->isInStack==FALSE)

{

for(node=lclause->clause->rmv;node&&cnf->varfloor[abs(node->var)]==floor;node=lclause->clause->rmv)

backvar(lclause->clause,node);

if(node==NULL)

lclause->isInStack=TRUE;

}

lclause=lclause->next;

}

return OK;

}

status deleteRepeateLearnClause(CNF\* cnf,LearnClause\* lclause)

{

Node\* nodep,\*nodeq;

LearnClause\* lcp=cnf->learn\_root;

lcp=lclause->next;

if(lcp)

{

nodep=lcp->clause->head;

if(nodep&&nodep->var==lclause->clause->head->var)

{

for(nodep=lcp->clause->rmv,nodeq=cnf->learn\_root->clause->rmv;nodep&&nodeq;nodep=nodep->next,nodeq=nodeq->next)

if(nodep->var!=nodeq->var)

break;

if(nodep==NULL&&nodeq==NULL) //重复学习子句

{

LearnClause\* temp=lcp->next;

deleteLearnClause(cnf,lcp);

lcp=temp;

}

else

lcp=lcp->next;

}

}

return OK;

}

status backChange(CNF\* cnf,ChangeStack\* head,int time,int\* learnarray)

{

//learnarray数组中，第0个元素保存了该数组的长度，第一个元素到第i个元素为冲突的变元，i表示撤销的次数，第i+1个元素为唯一蕴含点

//在蕴含图中，唯一蕴含点即为当前节点到冲突节点的所有路径中都要经过的节点

//唯一蕴含点所对应的变量的赋值是当前决策层产生冲突的直接原因

//学习子句就是将这些节点所对应的变元赋值取其返构成的字句加入子句集中

ChangeStack\* stack=head->next;

int flag=FALSE;

int\* IGraph; //蕴含图

int\* visitedIGraph; //变元是否赋值

int i=1,j;

int islearn=FALSE;

int backtofloor=head->floor;

Node\* node;

while(stack&&time!=0)

{

switch(stack->tag)

{

case CLAUSE:

insertClause(cnf,stack->Cp); //将被删除的子句放回子句集中

break;

case LITERAL:

backvar(stack->Cp,stack->Np); //恢复变元

if(learnarray&&flag==FALSE) //将产生冲突的变元加入a[]中

{

if(stack->Cp->length==1) //将发生冲突的变元加入到数组中

{

flag=TRUE;

IGraph=(int\* )malloc(sizeof(int)\*(cnf->varnum+2));

visitedIGraph=(int\* )malloc(sizeof(int)\*(cnf->varnum+1));

for(j=1;j<=cnf->varnum;j++)

visitedIGraph[j]=FALSE;

learnarray[i++]=stack->Cp->head->var; //存放该变元的变量

learnarray[i++]=-(stack->Cp->head->var);

visitedIGraph[abs(stack->Cp->head->var)]=TRUE; //变元var已经放入

}

}

break;

case SPLIT:

deleteClause(cnf,stack->Cp); //因为split是决策变量作为单子句插入进去的，所以恢复是应该删除

break;

default:

break;

}

if(learnarray&&islearn==FALSE)

{

if(stack->Cp->length==1)

{

learnarray[i]=-(stack->Cp->head->var); //监视哨，取负是因为stack里面的文字都是决策变元的非

for(j=1;stack->Cp->head->var!=-learnarray[j];j++)

;

if(j!=i)

{ //不存在，访问

IGraph[j]=TRUE;

int s=j;

for(j=2;j<i;j++)

if(IGraph[j]!=TRUE)

break;

if(i==j&&i!=3&&islearn==FALSE)

{

islearn=TRUE;

learnarray[0]=i-1; //下标为0存放冲突变元的个数

learnarray[i]=-learnarray[s];

int f=createLearnClause(cnf,learnarray+1,learnarray[0],-learnarray[s]);

if(f<backtofloor)

backtofloor=f;

}

else

{

for(node=stack->Cp->rmv;node;node=node->next) //删除重复的变元

{

if(visitedIGraph[abs(node->var)]==FALSE)

{

learnarray[i]=node->var;

IGraph[i]=FALSE;

visitedIGraph[abs(node->var)]=TRUE;

if(cnf->varfloor[abs(learnarray[i])]<head->floor)

IGraph[i]=TRUE;

i++;

}

}//end of for

}//end of else

}//end of if

}//end of if

}//end of if

time--;

head->next=head->next->next;

free(stack);

stack=head->next;

}//end of while

if(learnarray)

{

free(IGraph);

free(visitedIGraph);

learnarray[0]=backtofloor;

}

return OK;

}//end of backChange

int compare\_Des(const void \*a,const void \*b)

{

return ((Floor\*)b)->floor - ((Floor\*)a)->floor;

}

status check(CNF\* cnf)

{

Clause\* clause;

Node\* node;

int i,ans;

int flag=0;

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

printf("检查中\n");

for(clause=cnf->root,i=1;clause;clause=clause->next,i++)

{

printf("正在检查第%d个子句：",i);

for(node=clause->head;node;node=node->next)

{

ans=cnf->var[abs(node->var)];

if(EVALUE(node->var)==ans)

{

printf("正确! ");

flag=1;

break;

}

else if(EVALUE(node->var)==-ans)

continue;

else if(ans==NOTSURE)

printf("%d不确定(未出现) ",abs(node->var));

}

if(!flag)

printf("子句不满足！\n"); //只要有一个变元取值为真，则子句就是可满足的

if(i%4==0)

printf("\n");

else printf("\t\t");

}

printf("检查完成！\n");

return OK;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*cnfile.h

#ifndef CNFFILE\_H\_INCLUDED

#define CNFFILE\_H\_INCLUDED

#include "clause.h"

#define LITERALMAX 200

typedef int status;

CNF\* loadCNF(); //读取cnf文件

status changetoRes(char\* path,char\* cnfpath ); //求解结果输出到同名res文件

status saveRes(char\* respath,status dpll,CNF\* cnf,int time); //保存res文件

status saveCNF(); //保存cnf文件

#endif // CNFFILE\_H\_INCLUDED

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

cnfile.c

#include "cnfFile.h"

#include<string.h>

CNF\* loadCNF(char\* filename)

{

FILE\* fp;

char ch;

int var\_num,clause\_num;

fp=fopen(filename,"r");

int clause[200];

if(!fp)

{

printf("打开文件失败！\n");

exit(-1);

}

CNF\* cnf=(CNF\*)malloc(sizeof(CNF));

if(!cnf)

return NULL;

//读取注释

while((ch=fgetc(fp))=='c')

while((ch=fgetc(fp))!='\n')

;

for(int i=0;i<5;i++) //包括p cnf五个字符

ch=fgetc(fp);

fscanf(fp,"%d",&var\_num);

fscanf(fp,"%d",&clause\_num);

if(!createCNF(cnf,var\_num,0)) //这里不能直接将clause\_num创建，因为在create 函数中并未为子句创建空间

{

fclose(fp);

return NULL;

}

for(int i=0;i<clause\_num;i++)

{

int j=0;

do

{

fscanf(fp,"%d",&clause[j]);

j++;

}while(clause[j-1]!=0); //结束时前面j自增了，所以是j-1，是0，不是'0'

if(!addClause(cnf,j-1,clause))

{

fclose(fp);

return NULL;

}

}

fclose(fp);

return cnf;

}

status changetoRes(char\* resfilename,char\* cnffilename)

{

int i;

for(i=0;i<strlen(cnffilename)-4;i++) //开始写的-3，发现.无法放入resfilename中

resfilename[i]=cnffilename[i];

//此时res文件名为\*\*\*

resfilename[i]='\0';

strncat(resfilename,".res",100); //添加文件后缀

printf("res文件：%s\n",resfilename);

return OK;

}

status saveRes(char\* resfilename,status dpll,CNF\* cnf,int time)

{

//s为求解结果，v表示每个变元的取值，t表示时间

//传入的参数分别表示：待保存的res文件名，cnf文件，求解状态，花费时间

FILE\* fp=fopen(resfilename,"w");

if(!fp)

{

printf("文件打开失败！\n");

return ERROR;

}

fprintf(fp,"s %d\n",dpll);

fprintf(fp,"v");

for(int i=1;i<=cnf->varnum;i++)

fprintf(fp," %d",i\*(cnf->var[i]));

fprintf(fp,"\nt %d",time);

fclose(fp);

return OK;

}

status saveCNF(CNF\* cnf,char\* cnffilename)

{

FILE\* fp=fopen(cnffilename,"w");

Clause\* clause;

Node\* node;

if(!fp)

{

printf("文件打开失败!\n");

exit(-1);

}

fprintf(fp,"p cnf %d %d\n",cnf->varnum,cnf->clausenum);

for(clause=cnf->root;clause;clause=clause->next)

{

for(node=clause->head;node;node=node->next)

fprintf(fp,"%d ",node->var);

fprintf(fp,"0\n");

}

fclose(fp);

return OK;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sudoku.h

#ifndef SUDOKU\_H\_INCLUDED

#define SUDOKU\_H\_INCLUDED

#include "clause.h"

typedef struct Sudoku{

int sdk[6][6];

struct Sudoku\* next; //若干s数独有解，s.next指向其解

}Sudoku;

Sudoku\* createSudoku();

CNF\* transtoCNF(); //转换为cnf文件

status solveSudoku(); //求解数独

status DFSSSudoku();

status printSudoku(); //打印数独

Sudoku\* transtoSudoku(); //cnf转换为数独

status randSudoku(); //随机生成数独

status rule2\_select4unit(); //约束2，找任意四个单元

//添加到cnf文件中时子句转换的约束

status rule1();

status rule2();

status rule3();

//生成数独时的约束

status checkRule1();

status checkRule2();

status checkRule3();

status checkSudoku(); //生成数独填数是检查是否满足约束

status SaveSudoku(); //保存输入的数独到文件中

Sudoku\* loadSudoku(); //读取文件中的数独文件

status SolveMySukodu(); //验证输入的数独是否满足数独约束规则

status istheSameSudoku(); //检查输入的数独是否是待求解的数独

#endif // SUDOKU\_H\_INCLUDED

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sudoku.c

#include "sudoku.h"

#include "dpll.h"

Sudoku\* createSudoku(int difficult)

{

int i,j,k;

int x,y;

int flag=0;

Sudoku\* s=(Sudoku\* )malloc(sizeof(Sudoku));

Sudoku\* ss=NULL;

if(!s)

{

printf("内存分配不到！\n");

return NULL;

}

for(i=0;i<6;i++)

for(j=0;j<6;j++)

s->sdk[i][j]=-1;

if(randSudoku(s,0,0)==TRUE)

{

printf("\n\n数独终盘完成！-------------------------------------------\n\n");

//先随机挖掉一个空，此时生成的数独一定是只有一个解的

x=rand()%6;

y=rand()%6;

s->sdk[x][y]=-1;

//将数独盘大致均分为5个区间，对应到5个难度等级，每个区间大致为7-8个格子

int max=7\*difficult; //定义可以挖的洞的最多个数

while(1)

{

solveSudoku(s,&i); //求解数独，并获得解的个数i

if(i>1)

{

flag--;

s->sdk[x][y]=k; //新挖掉的s->sdk[x][y]格挖掉之后无法保证是唯一解，恢复这个格子原来的值

for(ss=s->next;ss;ss=s->next) //将之前求出的所有不唯一的解全部删除

{

s->next=ss->next;

free(ss);

}

}

else //第一次挖完一个之后，得到唯一解的数独盘，尝试继续挖洞

{

flag++;

if(flag>=max)

break;

do

{

x=rand()%6;

y=rand()%6;

}while(s->sdk[x][y]==-1); //找到可以挖的格子

k=s->sdk[x][y];

s->sdk[x][y]=-1; //挖洞

}

}

free(ss);

s->next=NULL;

return s;

}

else

return NULL;

}

status randSudoku(Sudoku\* s,int x,int y)

{

int i=-1;

int tmp=-2; //为了防止长时间随机赋值都赋为同一个值

do

{

do

{

i++;

if(i>1)

{

s->sdk[x][y]=-1;

return FALSE; //无论是0还是1都不能填入

}

tmp=s->sdk[x][y];

s->sdk[x][y]=rand()%2; //生成0-1

if(tmp==s->sdk[x][y])

s->sdk[x][y]=(s->sdk[x][y])?0:1; //更换

}while(checkSudoku(s)!=TRUE);

//printf("第%d行第%d列生成成功！\n",x,y);

if(x==5&&y==5)

{

printf("生成成功！\n");

return TRUE;

}

}while(((y==5)?randSudoku(s,x+1,0):randSudoku(s,x,y+1))!=TRUE);

return TRUE;

}

status SaveSudoku(void)

{

int s[6][6];

FILE\* fp;

char path[200];

printf("输入你的数独！\n用-1表示空格，其余填0或1\n");

for(int i=0;i<6;i++)

for(int j=0;j<6;j++)

scanf("%d",&s[i][j]);

printf("输入保存的路径！\n");

scanf("%s",path);

if((fp=fopen(path,"w+"))==NULL)

{

printf("文件保存失败！\n");

exit(-1);

}

for(int i=0;i<6;i++)

for(int j=0;j<6;j++)

fwrite(&s[i][j],sizeof(int),1,fp);

fclose(fp);

return OK;

}

Sudoku\* loadSudoku(void)

{

Sudoku\* s=(Sudoku\* )malloc(sizeof(Sudoku));

s->next=NULL; //开始无法确定解，记得赋值为NULL

printf("输入加载的文件名称：\n");

char filename[50];

scanf("%s",filename);

FILE\* fp=fopen(filename,"rb");

if(!fp)

{

printf("文件打开失败！\n");

return NULL;

}

for(int i=0;i<6;i++)

for(int j=0;j<6;j++)

fread(&s->sdk[i][j],sizeof(int),1,fp);

fclose(fp);

return s;

}

CNF\* transtoCNF(Sudoku\* s)

{

int i,j,k;

CNF\* cnf=(CNF\* )malloc(sizeof(CNF));

if(!createCNF(cnf,30000,0))

return NULL;

//将预填的格子的数字转化为子句加入到子句集中

for(i=0;i<6;i++)

{

for(j=0;j<6;j++)

{

k=(i+1)\*10+j+1;

if(s->sdk[i][j]==0)

{

k=-k;

addClause(cnf,1,&k);

}

else if(s->sdk[i][j]==1)

addClause(cnf,1,&k);

}

}

//printf("加入原数据成功！\n");

rule1(cnf);

//printf("约束一完成！\n");

rule2(cnf);

//printf("约束二完成！\n");

rule3(cnf);

//printf("约束三完成！\n");

return cnf;

}

status rule1(CNF\* cnf)

{

//约束一：每一行，每一列无连续的3个1或0

//每一行没有连续的3个1或0

int i,j,k;

int pos1[3];

int neg1[3]; //分别存放加入的三个连续单元正，负变量

int count;

for(i=0;i<6;i++)

{

for(j=0,count=0;j<6;j++,count++)

{

k=(i+1)\*10+j+1;

pos1[count%3]=k;

neg1[count%3]=-k;

if((count+1)%3==0) //每取得连续三个变量时，，就要增添子句

{

addClause(cnf,3,pos1);

addClause(cnf,3,neg1);

j -=2;

}

}

}

//每一列没有，互换i,j即可

for(j=0;j<6;j++)

{

for(i=0,count=0;i<6;i++,count++)

{

k=(i+1)\*10+j+1;

pos1[count%3]=k;

neg1[count%3]=-k;

if((count+1)%3==0) //每取得连续三个变量时，，就要增添子句

{

addClause(cnf,3,pos1);

addClause(cnf,3,neg1);

i -=2;

}

}

}

return OK;

}

status rule2(CNF\* cnf)

{

//约束二，每一行每一列中1和0的个数相同

//在6阶中，则1，0个数为3个，意味着任选的四个单元至少有一个1或者0

//对于每一行或者每一列有一共15中选法

int i,j;

int select[15][4]; //存放每一种可能

rule2\_select4unit(select); //只需将第一行的15中可能全部列出来，其余行，列均可由此变化

int pos\_select[15][4];

int neg\_select[15][4];

int count;

for(i=0;i<15;i++)

for(j=0;j<4;j++)

{

pos\_select[i][j]=select[i][j];

neg\_select[i][j]=-pos\_select[i][j];

}

//首先考虑行

for(i=0;i<6;i++)

{

for(count=0;count<15;count++)

{

addClause(cnf,4,pos\_select[count]);

addClause(cnf,4,neg\_select[count]);

}

for(count=0;count<15;count++)

for(j=0;j<4;j++)

{

pos\_select[count][j] +=10;

neg\_select[count][j]=-pos\_select[count][j];

}

}

//列的话其实就是将数组里面的数字进行反转

for(i=0;i<15;i++)

for(j=0;j<4;j++)

{

pos\_select[i][j]=(select[i][j])%10\*10+(select[i][j])/10;

neg\_select[i][j]=-pos\_select[i][j];

}

for(j=0;j<6;j++)

{

for(count=0;count<15;count++)

{

addClause(cnf,4,pos\_select[count]);

addClause(cnf,4,neg\_select[count]);

}

for(count=0;count<15;count++)

for(i=0;i<4;i++)

{

pos\_select[count][i] +=1; //注意这里，同行每一列之间只相差1

neg\_select[count][i]=-pos\_select[count][i];

}

}

return OK;

}

status rule3(CNF\* cnf)

{

//约束三，不存在重复的行和列

//任意两行不重复，需要引入1+6+6\*2=19个附加变元

//表示约束三供需要19\*15\*2=570个附加变元

//共添加2\*15\*（9\*6+7）=1830条附加子句

//1. 加入第x行第j列和第y行第j列均取1的子句，即1xyj1,以第3行和第5行为例

int i,j,x,y;

int line0[6]; //13510，13520...13560

int line1[6]; //13511，13521...13561

int line[6]; //1351，1352...1356

int var1,var2; //记录检查的两个单元

for(x=0;x<6;x++)

for(y=x+1;y<6;y++)

{

//将line0赋值并转换为子句加入子句集

for(j=0;j<6;j++) //第j个单元

{

var1=(x+1)\*10+j+1; //31

var2=(y+1)\*10+j+1; //51

line0[j]=0+(j+1)\*10+(y+1)\*100+(x+1)\*1000+1\*10000; //13510

line1[j]=1+(j+1)\*10+(y+1)\*100+(x+1)\*1000+1\*10000; //13511

//13511=31并51,产生三个子句,13511和13510共加入6条子句

int clause11[2],clause01[2];

clause11[0]=var1;clause11[1]=-line1[j];

clause01[0]=-var1;clause01[1]=-line0[j];

addClause(cnf,2,clause11);addClause(cnf,2,clause01);

int clause12[2],clause02[2];

clause12[0]=var2;clause12[1]=-line1[j];

clause02[0]=-var2;clause02[1]=-line0[j];

addClause(cnf,2,clause12);addClause(cnf,2,clause02);

int clause13[3],clause03[3];

clause13[0]=-var1;clause13[1]=-var2;clause13[2]=line1[j];

clause03[0]=var1;clause03[1]=var2;clause03[2]=line0[j];

addClause(cnf,3,clause13);addClause(cnf,3,clause03);

line[j]=j+1+(y+1)\*10+(x+1)\*100+1000; //1351

//1351转换为3条子句

int c1[2],c2[2],c3[3];

c1[0]=-line1[j];c1[1]=line[j];

addClause(cnf,2,c1);

c2[0]=-line0[j];c2[1]=line[j];

addClause(cnf,2,c2);

c3[0]=line0[j];c3[1]=line1[j];c3[2]=-line[j];

addClause(cnf,3,c3);

}

//135化为范式产生7条子句,第一条有7个变元，后六条均为2个变元，全部加入

int clause1[7];

int clause[6][2];

for(i=0;i<6;i++)

{

clause[i][0]=line[i]; //135i

clause1[i]=-line[i];

clause[i][1]=y+1+(x+1)\*10+100; //135

addClause(cnf,2,clause[i]);

}

clause1[6]=-((y+1)+(x+1)\*10+100);

addClause(cnf,7,clause1);

clause1[6]=-clause1[6];

addClause(cnf,1,&clause1[6]); //要将单独的135加入！！！

}

//2. 无重复列，将第一位的1换成列标2，2xyj1表示第x列和第y列的第j行都取1；

for(x=0;x<6;x++)

for(y=x+1;y<6;y++)

{

//将line0赋值并转换为子句加入子句集

for(j=0;j<6;j++) //第j个单元

{

var1=(j+1)\*10+x+1; //31 //一定要注意此时j是作为行，所以 应该是j+1在乘以10

var2=(j+1)\*10+y+1; //51

line0[j]=0+(j+1)\*10+(y+1)\*100+(x+1)\*1000+2\*10000; //23510

line1[j]=1+(j+1)\*10+(y+1)\*100+(x+1)\*1000+2\*10000; //23511

//23511=31并51,产生三个子句,13511和13510共加入6条子句

int clause11[2],clause01[2];

clause11[0]=var1;clause11[1]=-line1[j];

clause01[0]=-var1;clause01[1]=-line0[j];

addClause(cnf,2,clause11);addClause(cnf,2,clause01);

int clause12[2],clause02[2];

clause12[0]=var2;clause12[1]=-line1[j];

clause02[0]=-var2;clause02[1]=-line0[j];

addClause(cnf,2,clause12);addClause(cnf,2,clause02);

int clause13[3],clause03[3];

clause13[0]=-var1;clause13[1]=-var2;clause13[2]=line1[j];

clause03[0]=var1;clause03[1]=var2;clause03[2]=line0[j];

addClause(cnf,3,clause13);addClause(cnf,3,clause03);

line[j]=j+1+(y+1)\*10+(x+1)\*100+2000; //2351

//1351转换为3条子句

int c1[2],c2[2],c3[3];

c1[0]=-line1[j];c1[1]=line[j];

addClause(cnf,2,c1);

c2[0]=-line0[j];c2[1]=line[j];

addClause(cnf,2,c2);

c3[0]=line0[j];c3[1]=line1[j];c3[2]=-line[j];

addClause(cnf,3,c3);

}

//135化为范式产生7条子句,第一条有7个变元，后六条均为2个变元，全部加入

int clause1[7];

int clause[6][2];

for(i=0;i<6;i++)

{

clause[i][0]=line[i];

clause1[i]=-line[i];

clause[i][1]=y+1+(x+1)\*10+200; //235

addClause(cnf,2,clause[i]);

}

clause1[6]=-((y+1)+(x+1)\*10+200);

addClause(cnf,7,clause1);

clause1[6]=-clause1[6];

addClause(cnf,1,&clause1[6]);

}

return OK;

}

Sudoku\* transtoSudoku(CNF\* cnf)

{

Sudoku\* s=(Sudoku\* )malloc(sizeof(Sudoku));

s->next=NULL;

int i,j;

for(i=0;i<6;i++)

for(j=0;j<6;j++)

{

int var=j+1+(i+1)\*10;

//locvar(cnf,var);

if(cnf->var[var]==TRUE)

s->sdk[i][j]=1;

else

s->sdk[i][j]=0;

}

return s;

}

status rule2\_select4unit(int s[][4])

{

int i;

//首先搞定第一个

for(i=0;i<10;i++) //11开头有十种

s[i][0]=11;

for( ;i<14;i++) //12开头有四种,i从10开始，10，11，12，13均为12开头

s[i][0]=12;

s[i][0]=13;

//第二个

for(i=0;i<6;i++) //第二个为12的有6个

s[i][1]=12;

for( ;i<9;i++) //第二个为13的有6，7，8

s[i][1]=13;

s[i][1]=14; //第二个为14的仅一个，i=9;

for(i=10;i<13;i++) //i=10,11,12取13

s[i][1]=13;

for( ;i<15;i++) //i=13,14取14

s[i][1]=14;

//第三个

s[0][2]=s[1][2]=s[2][2]=13;

s[3][2]=s[4][2]=s[6][2]=s[7][2]=s[10][2]=s[11][2]=14;

s[5][2]=s[8][2]=s[9][2]=s[13][2]=s[14][2]=s[12][2]=15;

//第四个

s[0][3]=14;

s[1][3]=s[3][3]=s[6][3]=s[10][3]=15;

s[2][3]=s[4][3]=s[5][3]=s[7][3]=s[8][3]=s[9][3]=s[11][3]=s[12][3]=s[13][3]=s[14][3]=16;

return OK;

}

status checkSudoku(Sudoku\* s)

{

if(checkRule1(s)==TRUE)

{

//printf("约束一满足！\n");

if(checkRule2(s)==TRUE)

{

//printf("约束二满足！\n");

if(checkRule3(s)==TRUE)

{

// printf("约束三满足！\n");

return TRUE;

}

else

{

// printf("约束三不满足！\n");

return FALSE;

}

}

else

{

// printf("约束二不满足！\n");

return FALSE;

}

}

else

{

// printf("约束一不满足！\n");

return FALSE;

}

}

status checkRule1(Sudoku\* s)

{

//检查数独是否合乎约束

//约束一

//检查行

int i,j;

int count;

for(i=0;i<6;i++)

{

for(j=0,count=0;j<6;j++,count++)

{

if(!(s->sdk[i][j]==0||s->sdk[i][j]==1||s->sdk[i][j]==-1))

return FALSE;

else

{

if(s->sdk[i][j]==0&&s->sdk[i][j+1]==0&&s->sdk[i][j+2]==0)

{

//printf("第%d %d %d列第%d行不满足！\n",j,j+1,j+2,i);

return FALSE;

}

else if(s->sdk[i][j]==1&&s->sdk[i][j+1]==1&&s->sdk[i][j+2]==1)

{

// printf("第%d %d %d列第%d行不满足！\n",j,j+1,j+2,i);

return FALSE;

}

else

{

if((count+1)%3==0&&count<11)

j-=2;

else

continue;

}

}

}

}

//检查列

for(j=0;j<6;j++)

{

for(i=0,count=0;i<6;i++,count++)

{

if(!(s->sdk[i][j]==0||s->sdk[i][j]==1||s->sdk[i][j]==-1))

{

return FALSE;

}

else

{

if(s->sdk[i][j]==0&&s->sdk[i+1][j]==0&&s->sdk[i+2][j]==0)

{

// printf("第%d %d %d行第%d列不满足!\n",i,i+1,i+2,j);

return FALSE;

}

else if(s->sdk[i][j]==1&&s->sdk[i+1][j]==1&&s->sdk[i+2][j]==1)

{

// printf("第%d %d %d行第%d列不满足!\n",i,i+1,i+2,j);

return FALSE;

}

else

{

if((count+1)%3==0&&count<11)

i-=2;

else

continue;

}

}

}

}

return TRUE;

}

status checkRule2(Sudoku\* s)

{

//约束二

int i,j;

int count0,count1;

//检查行

for(i=0;i<6;i++)

{

for(j=0,count0=0,count1=0;j<6;j++)

{

if(s->sdk[i][j]==0)

count0++;

else if(s->sdk[i][j]==1)

count1++;

}

//if(count0!=count1&&count0+count1==6)

if(abs(count0-count1)>2||count0+count1==6)

if(!(count0==count1&&count0==3))

return FALSE;

}

//检查列

for(j=0;j<6;j++)

{

for(i=0,count0=0,count1=0;i<6;i++)

{

if(s->sdk[i][j]==0)

count0++;

else if(s->sdk[i][j]==1)

count1++;

}

// if(count0!=count1&&count0+count1==6)

if(abs(count0-count1)>2||count0+count1==6) //abs句保证不会出现4个1，2个0的情况

if(!(count0==count1&&count0==3))

return FALSE;

}

return TRUE;

}

status checkRule3(Sudoku\* s)

{

int i,j,k;

int count;

//检查行

for(i=0;i<6;i++) //第i行和第j行比较

{

for(j=i+1,count=0;j<6;j++)

{

for(k=0;k<6;k++)

{

if(s->sdk[i][k]==s->sdk[j][k]&&s->sdk[i][k]!=-1&&s->sdk[j][k]!=-1)

count++;

}

if(count==6)

return FALSE;

count=0; //和j比完和j+1比较后记得将count清零

}

}

//检查列

for(j=0;j<6;j++) //第j列和第i列比较

{

for(i=j+1,count=0;i<6;i++)

{

for(k=0;k<6;k++)

{

if(s->sdk[k][j]==s->sdk[k][i]&&s->sdk[k][j]!=-1&&s->sdk[k][i]!=-1)

count++;

}

if(count==6) //重复的格子数为满格且不是均为-1

return FALSE;

count=0;

}

}

return TRUE;

}

status solveSudoku(Sudoku\* s,int\* solvecount)

{

\*solvecount=0;

DFSSSudoku(s,0,0,solvecount);

return (solvecount>0)?TRUE:FALSE;

}

status DFSSSudoku(Sudoku\* s,int x,int y,int\* solvecount)

{

//深度优先的方法找到所有解

int i,j,k;

Sudoku\* ss;

while(s->sdk[x][y]!=-1&&x<=5) //找到被挖掉的格子

{

y++;

if(y==6)

{

x++; //换行

y=0;

}

}

if(x>5) //已经全部填好

{

ss=(Sudoku\* )malloc(sizeof(Sudoku));

for(j=0;j<6;j++)

for(k=0;k<6;k++)

ss->sdk[j][k]=s->sdk[j][k];

ss->next=s->next;

s->next=ss;

(\*solvecount)++;

return TRUE;

}

//if(x>5)

//return TRUE;

for(i=0;i<=1;i++)

{

s->sdk[x][y]=i; //尝试填数

switch(checkSudoku(s)) //每次尝试检查是否满足数独游戏的三个约束

{

case FALSE:

break;

case TRUE:

if(x==5&&y==5) //数独全部填完

{

ss=(Sudoku\*)malloc(sizeof(Sudoku));

for(j=0;j<6;j++)

for(k=0;k<6;k++)

ss->sdk[j][k]=s->sdk[j][k];

ss->next=s->next;

s->next=ss;

s->sdk[x][y]=-1;

(\*solvecount)++;

return TRUE;

}

else

(y==5)?DFSSSudoku(s,x+1,0,solvecount):DFSSSudoku(s,x,y+1,solvecount); //填完一行从下一行的第一个格子开始，否则直接接右边的格子

break;

default:

break;

}

}

s->sdk[x][y]=-1; //求得值后恢复为待填状态

return FALSE;

}

status printSudoku(Sudoku\* s,status color)

{

//边框打印白色，数字打印绿色

//当color取false是为数字

int i,j;

printf("打印数独！\n");

printf("\t ");

for(j=0;j<11;j++)

printf("-");

printf("\n");

for(i=0;i<6;i++)

{

printf("\t");

printf("|");

for(j=0;j<6;j++)

{

if(s->next)

{

if(s->sdk[i][j]==-1)

printf("%d%c",s->next->sdk[i][j],(j+1)%3==0?'|':' ');

else

//字的背景颜色是黑色，字体颜色是绿色

printf("\033[40;32m%d\033[0m%c",s->sdk[i][j],((j+1)%3==0)?'|':' ');

}

else

{

if(s->sdk[i][j]==-1)

printf(" %c",((j+1)%3==0)?'|':' ');

else if(color==FALSE)

printf("%d%c",s->sdk[i][j],((j+1)%3==0)?'|':' ');

else

printf("\033[40;32m%d\033[0m%c",s->sdk[i][j],((j+1)%3==0)?'|':' ');

}

}

printf("\n");

if((i+1)%3==0)

{

printf("\t");

printf("|");

for(j=0;j<11;j++)

printf("-");

printf("|\n");

}

}

return OK;

}

status SolveMySukodu(Sudoku\* ss,Sudoku\* s)

{

//ss是待求解的数独盘，s是待输入的数独盘

s->next=NULL;

printf("请按照盘局输入你的答案\n");

for(int i=0;i<6;i++)

for(int j=0;j<6;j++)

scanf("%d",&(s->sdk)[i][j]);

if(istheSameSudoku(ss,s)==TRUE)

{

if(checkSudoku(s)==TRUE)

{

printf("求解成功！");

return OK;

}

}

else

printf("输入的数独不是求解的数独！\n");

printf("求解失败！\n");

printf("1. 再次求解 2.求助系统\n");

int op2;

scanf("%d",&op2);

switch(op2)

{

case 1:

SolveMySukodu(ss,s);

break;

case 2:

return FALSE;

}

}

status istheSameSudoku(Sudoku\* ss,Sudoku\* s)

{

//检查生成的数独盘和输入求解的数独盘是否相同

//主要是对未挖单元格进行检查即对生成数独盘的非-1格对应的是否相同

for(int i=0;i<6;i++)

for(int j=0;j<6;j++)

{

if((ss->sdk)[i][j]!=-1)

if((ss->sdk)[i][j]==(s->sdk)[i][j])

;

else

return FALSE;

}

return TRUE;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

display.h

#ifndef DISPLAY\_H\_INCLUDED

#define DISPLAY\_H\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "cnfFile.h"

#include "DPLL.h"

#include "sudoku.h"

#include <io.h>

#include <string.h>

void CNFpage();

void SUDOKUpage();

#endif // DISPLAY\_H\_INCLUDED

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*display.c

#include "display.h"

#define DEBUG 0

void CNFpage()

{

int op=1;

CNF\* cnf=NULL;

char cnffile[200]; //注意因为一个汉子占2个字节，而保存的用例的路径含有较多汉字，导致之前写100不够

static int flag=FALSE; //flag用来表示是否已经载入文件

int t; //

int dpll;

char res\_save\_path[200];

while(op)

{

system("cls");

printf("\n\n");

printf(" Menu for CNF based on SAT \n");

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

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

printf("1. 载入cnf文件 2. 求解cnf文件\n");

printf("3. 打印cnf文件 4. 检查 \n");

printf("0. 返回上一级\n");

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

printf("选择你的操作【0-3】：");

scanf("%d",&op);

switch(op)

{

case 1:

printf("输入加载的文件的路径：\n");

scanf("%s",cnffile);

if(strcmp(cnffile+strlen(cnffile)-4,".cnf")==0) //确保是cnf文件

{

if(!(cnf=loadCNF(cnffile)))

printf("文件打开失败！\n");

else

printf("打开成功！\n");

}

flag=TRUE;

getchar();getchar();

break;

case 2:

if(flag==FALSE)

printf("尚未加载cnf文件！\n");

else if(!cnf)

printf("cnf不存在!\n");

else

{

int type;

clearCNF(cnf);

printf("该cnf文件的变元数为：%d\n",cnf->varnum);

printf("该cnf文件的子句数为：%d\n",cnf->clausenum);

printf("选择： 1. 优化求解 2.正常求解\n");

scanf("%d",&type);

switch(type)

{

case 1:

createLIndex(cnf);

t=clock();

dpll=advancedDPLL(cnf,0);

t=clock()-t;

break;

case 2:

t=clock();

dpll=DPLL(cnf,0);

t=clock()-t;

break;

default:

break;

}//end of awitch

if(dpll==TRUE)

{

printf("求解成功！\n");

printf("求解答案：\n");

for(int i=1;i<=cnf->varnum;i++) //如果输出为0，则表示这个变元在cnf文件中没有出现

{

printf("%d ",cnf->var[i]\*i);

if(i%10==0)

printf("\n");

}

printf("\n");

printf("保存res文件\n");

changetoRes(res\_save\_path,cnffile);

if(saveRes(res\_save\_path,dpll,cnf,t))

printf("res文件保存成功！\n");

else

printf("res文件保存失败！\n");

}

else

{

printf("求解失败！\n");

printf("保存res文件\n");

changetoRes(res\_save\_path,cnffile);

if(saveRes(res\_save\_path,dpll,cnf,t))

printf("res文件保存成功！\n");

else

printf("res文件保存失败！\n");

}

printf("耗费时间：%dms\n",t);

}//end of else

getchar();getchar();

break;

case 3:

printf("打印cnf文件:\n");

if(!cnf)

printf("cnf文件不存在！\n");

else

printClause(cnf);

getchar();getchar();

break;

case 4:

if(!cnf)

printf("文件不存在！\n");

else if(dpll==FALSE)

printf("求解错误！\n");

else

check(cnf);

getchar();getchar();

break;

case 0:

break;

}//end of switch

}//end of while(op)

}//end of cnfpage

status SudokuPage()

{

int op=1;

while(op)

{

printf("\n\n");

printf(" Menu for SUDOKU \n");

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

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

printf("1. 输入并保存你的数独 2. 读取输入的数独并求解\n");

printf("3. 求解随机数独 0. 退出 \n");

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

printf("选择你的操作【0-3】：");

scanf("%d",&op);

switch(op)

{

case 1:

SaveSudoku();

getchar();getchar();

break;

case 2:

{

int t;

int dpll;

Sudoku\* s=loadSudoku();

if (!s)

break;

for(int i=0;i<6;i++)

{

for(int j=0;j<6;j++)

printf("%d ",s->sdk[i][j]);

printf("\n");

}

printSudoku(s,TRUE);

printf("\n");

CNF\* cnf=NULL;

cnf=transtoCNF(s);

Clause\* clause;Node\* node;

int i=0;

for(clause=cnf->root;clause;clause=clause->next)

for(node=clause->head;node;node=node->next)

{

if(abs(node->var)>i)

i=abs(node->var);

}

printf("cnf文件中变元的最大值wei :%d\n",i);

if(!cnf)

{

printf("转换失败！\n");

return FALSE;

}

createLIndex(cnf);

t=clock();

dpll=advancedDPLL(cnf,0);

t=clock()-t;

if(dpll==TRUE)

{

printf("数独成功解出！\n");

printf("用时：%dms\n",t);

Sudoku\* ss=transtoSudoku(cnf);

s->next=ss;

printSudoku(s,FALSE);

free(ss);

}

else

printf("求解失败\n");

getchar();getchar();

break;

}

case 3:

{

CNF\* cnf;

int t,dpll;

Sudoku\* s;

int difficult;

printf("输入难度：[1-3]\n"); //难度表示计算允许多解的次数

scanf("%d",&difficult);

if(!(s=createSudoku(difficult)))

{

printf("数独生成失败！\n");

break;

}

else

{

printSudoku(s,TRUE);

int op1;

printf("1. 自己求解 2. 系统求解\n");

scanf("%d",&op1);

switch(op1)

{

case 1:

{

Sudoku\* s1=(Sudoku\* )malloc(sizeof(Sudoku));

if(!s1)

{

printf("创建数独失败！\n");

break;

}

int state=SolveMySukodu(s,s1);

if(state==OK)

break;

}

case 2:

{

cnf=transtoCNF(s);

if(!cnf)

{

printf("cnf文件转换失败！\n");

return ERROR;

}

else

{

printf("数独成功转换为cnf!\n");

printf("变元数为:%d\n子句数为:%d\n",cnf->varnum,cnf->clausenum);

printf("正在使用优化后的求解器为您求解...\n");

//createLIndex(cnf);

t=clock();

dpll=DPLL(cnf,0);

t=clock()-t;

if(dpll==TRUE)

{

printf("数独成功解出！\n");

printf("用时：%dms\n",t);

Sudoku\* ss=transtoSudoku(cnf);

s->next=ss;

printSudoku(s,FALSE);

free(ss);

while(1)

{

int op;

char cnffilename[200];

char resfilename[200];

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

printf("1. 保存cnf文件及结果 ");

printf("0.返回\n");

printf("你的选择：【0-1】：\n");

scanf("%d",&op);

if(op==1)

{

printf("输入你保存的路径:\n");

scanf("%s",cnffilename);

if(saveCNF(cnf,cnffilename))

{

printf("cnf保存成功！\n");

changetoRes(resfilename,cnffilename);

if(saveRes(resfilename,dpll,cnf,t))

{

printf("res文件保存成功！\n");

break;

}

else

printf("res文件保存失败！\n");

}

else

printf("cnf文件保存失败！\n");

}

else if(op==0)

break;

else

{

printf("输入错误！\n");

break;

}

}//end of while

}//end of if(dpll==TRUE)

else

printf("数独求解失败！\n");

}

}

}

}

getchar();getchar();

break;

}

case 0:

break;

}

}

return OK;

}