

**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： 计科校交1803**

**学 号： U201810824**

**姓 名： 孟彦康**

**指导教师： 周全**

**报告日期： 2020/2/1**

**计算机科学与技术学院**

# 任务书

## 1.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。

## 2.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。
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。
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。
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. **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。
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。

## 3.参考文献

[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]TanbirAhmed.An Implementation of the DPLL Algorithm.Masterthesis,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]CarstenSinz.Visualizing SAT Instances and Runsof the DPLL Algorithm.JAutom 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 forProgramming, Artificial Intelligence and Reasoning, LPAR 2005, pages 11–15, 2005.

[8]InsLynce and JolOuaknine. 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

**目录**

**[任务书 I](#_Toc23071)**

**[1.设计内容 I](#_Toc3752)**

**[2.设计要求 I](#_Toc23204)**

**[3.参考文献 II](#_Toc10647)**

**[基于SAT的二进制数独游戏求解程序 1](#_Toc5368)**

**[1 问题概述 1](#_Toc14168)**

**[2 DPLL算法思想 2](#_Toc6942)**

**[3 功能要求 3](#_Toc26057)**

**[4系统设计 4](#_Toc24511)**

**[4.1数据物理结构 4](#_Toc28718)**

**[4.2 演示系统 4](#_Toc24417)**

**[4.3 cnf文件读取和解析实现算法 5](#_Toc7673)**

**[4.4 二进制数独游戏实现算法 8](#_Toc13454)**

**[5 系统实现 9](#_Toc21735)**

**[5.1 实验环境 9](#_Toc30208)**

**[5.2 系统演示 9](#_Toc16630)**

**[5.3 系统测试 10](#_Toc30534)**

**[6 实验小结 13](#_Toc26867)**

**[附录 程序源代码 15](#_Toc1921)**

# 基于SAT的二进制数独游戏求解程序

## 1 问题概述

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。

对于任一布尔变元*x*，*x*与其非“*¬x*”称为文字(literal)。对于多个布尔变元，若干个文字的或运算*l*1∨*l*2∨…∨*lk*称为子句(clause)。只含一个文字的子句称为单子句。不含任何文字的子句称为空子句，常用符号□表示。子句所含文字越多，越易满足，空子句不可满足。

SAT问题一般可描述为：给定布尔变元集合{*x*1, *x*2, ..., *x*n}以及相应的子句集合{*c*1, *c*2, ..., *c*m}，对于合取范式（CNF范式）：*F* = *c*1∧*c*2∧...∧*c*m，判定是否存在对每个布尔变元的一组真值赋值使*F*为真，当为真时（问题是可满足的，SAT），输出对应的变元赋值（一组解）结果。

一个CNF公式也可以表示成子句集合的形式：*S*={*c*1,*c*2,...,*c*m}.

例如，由三个布尔变元a,b,c所形成的一个CNF公式（¬a∨b）∧（¬b∨c）,可用集合表示为{¬a∨b,¬b∨c}，该公式是满足的，a=0, b=0,c=1是其一组解。

一个CNF SAT公式或算例的具体信息通常存储在一个.cnf文件中，下图2.1是算例problem1.cnf文件前若干行的截图。



图1-1 cnf 文件格式

在每个CNF文件的开始，由‘c’开头的是若干注释说明行；‘p’开头的行说明公式的总体信息，包括：范式为CNF；公式有200个布尔变元，由1到200的整数表示；320个子句。之后每行对应一个子句，0为结束标记。46表示第46号变元，且为正文字；-46则是对应的负文字，文字之间以空格分隔。

DPLL算法是经典的SAT完备型求解算法，对给定的一个SAT问题实例，理论上可判定其是否满足，满足时可给出对应的一组解。

## 2 DPLL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.

交错使用上述两种策略可不断地对公式化简，并最终达到终止状态，其执行过程可表示为一棵二叉搜索树,如下图2.2所示。



图2-1 DPLL算法搜索树

基于单子句传播与分裂策略的DPLL算法可以描述为一个如后所示的递归过程**DPLL( *S* )**,为了优化执行效率，可用非递归实现。

**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**

**}**

对于公式{¬1∨2, ¬2，¬3∨4, 3∨¬5, 3∨4, 3∨5，¬2∨¬5∨6} ,大家可以利用DPLL算法进行手动推理其搜索处理及回溯过程，获得求解结果。

## 3 功能要求

本设计要求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，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。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。
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。
3. **DPLL过程：**基于DPLL算法框架，实现SAT算例的求解。
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。

**程序优化：**对基本DPLL的实现进行存储结构、分支变元选取策略等某一方面进行优化设计与实现，提供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。

**SAT应用：**将二进制数独游戏问题转化为SAT问题，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。

## 4系统设计

### 4.1数据物理结构

十字链表结构体定义如下：

typedef struct SATNode{

int data; //数据域

SATNode\* next;

}SATNode;

typedef struct SATList {

SATNode\* head; //表头

SATList\* next;

}SATList;



图1.4.1-1 十字链表结构示意图

### 4.2 演示系统

演示系统包括用户操作界面和功能调用部分。

演示系统界面和所有操作和提示语言为中文。

用户操作界面输出可选的功能操作，用户输入数字选择要进行的操作。在

用户选择操作后，功能调用部分会显示函数的名称，参数，返回值和作用，系统

提示用户输入参数。

功能调用部分将用户输入的有关信息传递给线性数据结构的操作函数进行调用，并对函数的返回值进行处理判断输出相应的提示信息。

### 4.3 cnf文件读取和解析实现算法

本部分说明算法时直接用函数原型代替。

1. **int ReadFile(SATList\*& cnf);**

**功能：**用文件指针fp打开用户指定的文件，并读取文件内容保存到给定参数中，读取成功返回1，失败返回0。

**算法实现：**依次读取文件内容，读取到cnf时，记录后面的变元数和句子数。再依次开辟十字链表空间，记录每一条句子和文字。

**时空效率分析：**依次遍历每一条句子的每个文字，时间复杂度为O(n\*m)，需给十字链表开辟空间，空间复杂度为O(n\*m)，n和m分别为句子数量和每个句子所含文字数量。

1. **void destroyClause(SATList\*& cnf);**

**功能：**销毁链表。

**算法实现：**设置记录变量，遍历销毁十字链表的每一句子和每一个文字。

**时空效率分析：**需遍历每一个句子和每一个文字，时间复杂度为O(n\*m)，空间复杂度为O(1)，n和m分别为句子数量和每个句子所含文字数量。

1. **int isUnitClause(SATNode\* cnf);**

**功能：**判断是否为单子句，是返回1，不是返回0。

**算法实现：**判断传入参数句子是否符合“头指针不为空，下一指针为空”的条件，符合即为单子句，否则不是单子句。

**时空效率分析：**时间复杂度和空间复杂度均为O(1)。

1. **int evaluateClause(SATNode\* cnf, int v[]);**

**功能：**评估子句的真假状态，真返回1，假返回0。

**算法实现：**根据结果记录数组v里面的值，如果参数句子cnf里的某一个文字真值与v中的对应，则该句子为真，否则为假。

**时空效率分析：**需遍历该句子的所有文字，时间复杂度为O(m)，空间复杂度为O(1)，m是句子中的文字数量。

1. **int removeClause(SATList\*& cnf, SATList\*& root);**

**功能：**在已有的十字链表中删除指定的子句，删除成功返回1，失败返回0。

**算法实现：**若删除为头，则使头指针指向下一个结点，然后释放内存；若删除为其他位置，找到该结点的直接前驱，使其next指针指向要删除结点的下一结点，然后释放删除结点内存。

**时空效率分析：**需遍历查找指定句子，时间复杂度为O(n)，空间复杂度为O(1)，n是链表中句子的数量。

1. **int removeNode(SATNode\*& cnf, SATNode\*& head);**

**功能：**在指定的子句中删除指定的文字，删除成功返回1，失败返回0。

**算法实现：**若删除为头，则使头指针指向下一个结点，然后释放内存；若删除为其他位置，找到该结点的直接前驱，使其next指针指向要删除结点的下一结点，然后释放删除结点内存。

**时空效率分析：**需遍历查找指定文字，时间复杂度为O(m)，空间复杂度为O(1)，m是句子中文字的数量。

1. **int addClause(SATList\* cnf, SATList\*& root);**

**功能：**在已有的十字链表中添加指定的子句，添加成功返回1，失败返回0。

**算法实现：**直接添加在已有链表头部，使该结点next指针指向原链表头，然后原链表头赋值为该节点。

**时空效率分析：**时间复杂度和空间复杂度均为O(1)。

1. **int emptyClause(SATList\* cnf);**

**功能：**判断是否含有空子句，是返回1，不是返回0。

**算法实现：**遍历所有句子，若出现某一句子不含任何文字即head为空，则含有空子句，否则不含。

**时空效率分析：**需遍历链表中所有句子，时间复杂度为O(n)，空间复杂度为O(1)，n是子句的数量。

1. **int DPLL(SATList\*& cnf, int value[]);**

**功能：**求解SAT问题，给出满足条件时的一个式子,若有解则返回1，无解返回0。

**算法实现：**

1）查找链表中的单子句tp，使结果记录数组v中对应记录为真，即v[tp->data]=1或v[-tp->data]=0。

2）删除所有含文字tp->data的子句，以及删除所有负文字-tp->data。

3）删除完后，判断链表是否为空，若为空，则说明问题已解决，返回1；判断链表中是否含有空子句，若有，则说明问题无解，返回0；

4）重复1）、2）、3)直到链表中不出现单子句。

5）选取变元a策略：记录每一个变元出现的次数（正文字和负文字区别开来），若链表中有正文字存在，则在正文字中找到出现次数最多的变元，该变元为所选变元。若链表中只有负文字，则在负文字中找到出现次数最多的变元，该变元为所选变元。

6）在原有链表中添加只含文字a的单子句，递归DPLL函数，从步骤1）开始继续运算，若结果为1，则问题求解成功，返回1；若结果为0，则在原有链表中添加只含文字-a的单子句，再次递归DPLL函数，并直接返回求解结果。

**时空效率分析：**需遍历每个句子查找单子句，且要遍历每个文字找到出现次数最多的变元，即每一步的时间复杂度为O(n\*m)，最坏情况下，总共需走2^b步，即总时间复杂度为O(n\*m\*2^b)，每一步要记录先前的链表，空间复杂度为O(b)，n是链表子句数量，m是句子中文字的数量，b是总变元的数量。

1. **void CopyClause(SATList\*& a, SATList\* b);**

**功能：**将链表b的值复制到链表a中。

**算法实现：**对链表a依次开辟空间，并将b中每个句子和文字赋值给a中对应的结点。

**时空效率分析：**需遍历b中每个句子和文字，时间复杂度为O(n\*m)，需对a依次开辟空间，空间复杂度为O(n\*m)，n和m分别为句子数量和句子中文字的数量

1. **int WriteFile(int result, double time, int value[]);**

**功能：**将运行结果保存至同名文件，文件拓展名为.res,保存成功返回1，失败返回0。

**算法实现：**修改fileName的后三个字母，使其为res。然后依次写入运行结果result、求解记录value和运行时间time。

**时空效率分析：**需遍历写入求解记录，时间复杂度为O(b)，空间复杂度为O(1)，b是变元数量。

### 4.4 二进制数独游戏实现算法

**1.void CreateBinary(int size);**

**功能：**按照指定大小生成棋盘。

**算法实现：**开辟一个size\*size+1大小的数组chess，然后在随机的一个位置生成一个1或0，再将棋盘chess通过SolvePuzzle()求解，得到一个满足二进制数独规则的棋局。随机选择size \* (size - 2)到size \* (size - 2)+(size / 2)个数字清空，然后打印棋盘。

**时空效率分析：**调用函数SolvePuzzle，开辟chess棋盘数组，时间复杂度同SolvePuzzle，空间复杂度为O(n\*n)，n是棋盘大小。

**2.int SolvePuzzle(int chess[], int size);**

**功能：**求解输入棋盘，有解返回1，无解返回0。

**算法实现：**根据棋盘已有数字和二进制数独满足的约束，建立cnf式子，再根据DPLL函数求解。

1. 添加单子句：若棋盘已有数字，为1添加对应正文字单子句，为0添加对应负文字单子句。
2. 约束1（不出现连续三个重复数字）：遍历所有行和列，每三个连续变元需满足a∨b∨c，¬a∨¬b∨¬c，即添加对应的两个句子。
3. 约束2（在每一行、每一列中1与0的个数相同）：遍历所有行和列，每一行（列）任选的size/2+1个变元中，必须不全为0或不全为1，即a∨b∨c∨d∨e，¬a∨¬b∨¬c∨¬d∨¬e，通过数组记录状压法，遍历所有组合情况，添加对应句子。
4. 约束3（不存在重复的行与重复的列）：以第5行与第7行为例，不能有完全相同的填充，则须满足：

¬{[（51∧71）∨（¬51∧¬71）]∧[（52∧72）∨（¬52∧¬72）] ∧…∧[（58∧78）∨（¬58∧¬78）]}，化简得到：

[（¬51∧71）∨（51∧¬71）]∨[（¬52∧72）∨（52∧¬72）] ∨…∨[（¬58∧78）∨（58∧¬78）]

引入新变元a和b使得a=¬51∧71，b=51∧¬71，依次类推，则可将其化简为一个长句a∨b∨c∨d.....

为满足变元连续，故每次引入新变元只需引入变元a=b+1，b为原来变元数量，并添加引入变元的三个条件句：¬51∨¬a；71∨¬a；51∨¬71∨a（举例添加a=¬51∧71）。

遍历所有行和列，按照引入变元法添加对应句子。

**时空效率分析：**调用函数DPLL，时间复杂度同DPLL，开辟链表写入cnf表达式，空间复杂度为O(n\*m)，n是句子数量，m是句子中文字数量。

## 5 系统实现

### 5.1 实验环境

实验环境为 Windows 10，代码采用编辑器 Visual Studio 2019 编写。

文件说明：

\* 数据结构课程设计.cpp：整个系统构建和用户操作实现

### 5.2 系统演示

程序主控流程：



图5.2-1 主控流程示意图

界面展示：

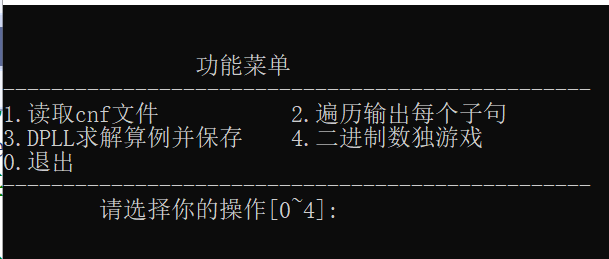


图5.2-2 界面展示示意图

### 5.3 系统测试

**1.cnf文件读取**（读取文件：ais12.cnf）：

读取结果：

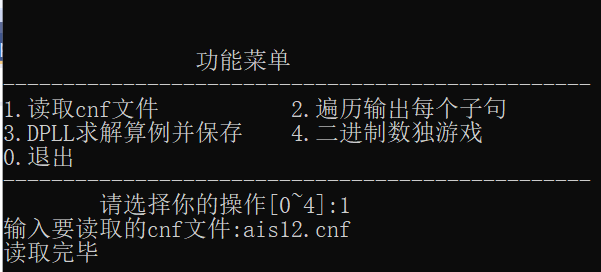


图5.3-1 读取文件

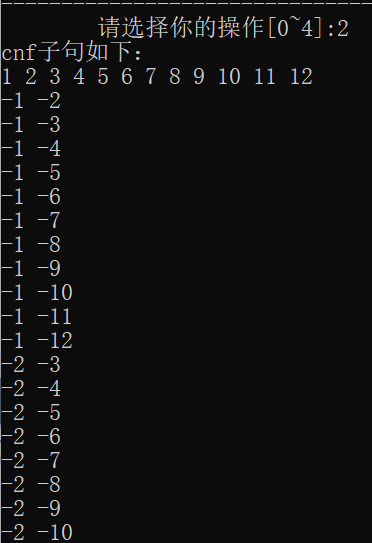
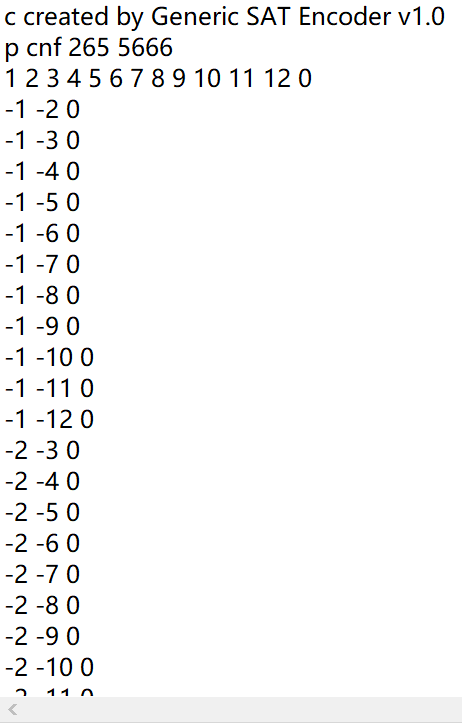
 

图5.3-2 读取文件部分内容对比

**2.DPLL求解算例**

|  |  |  |
| --- | --- | --- |
| 求解文件 | 求解结果 | 求解时间 |
| 7cnf20\_90000\_90000\_7.shuffled-20.cnf  （可满足算例S级） | 1 | 44ms |
| problem1-20.cnf  （可满足算例S级） | 1 | 1ms |
| problem2-50.cnf  （可满足算例S级） | 1 | 14ms |
| problem3-100.cnf  （可满足算例S级） | 1 | 2426ms |
| problem6-50.cnf  （可满足算例S级） | 1 | 204ms |
| sud00012.cnf  （可满足算例M级） | 1 | 213ms |
| sud00021.cnf  （可满足算例M级） | 1 | 1523ms |
| sud00079.cnf  （可满足算例M级） | 1 | 91ms |
| sud00082.cnf  （可满足算例M级） | 1 | 250.ms |
| sud00861.cnf  （可满足算例M级） | 1 | 46ms |
| bart17.shuffled-231.cnf  （可满足算例M级） | 1 | 50ms |
| sud00009.cnf  （可满足算例M级） | 1 | 207ms |
| eh-dp04s04.shuffled-1075.cnf  （可满足算例L级） | 1 | 7184ms |
| u-problem7-50.cnf  （不满足算例） | 0 | 785ms |
| tst\_v10\_c100.cnf  （不满足算例） | 0 | 0.0000ms |
| php-010-008.shuffled-as.sat05-1171.cnf  （不满足算例） | 0 | 13266ms |
| u-5cnf\_3500\_3500\_30f1.shuffled-30.cnf  （不满足算例） | 0 | 295ms |
| ais6.cnf | 1 | 3ms |
| ais8.cnf | 1 | 23ms |
| ais10.cnf | 1 | 96ms |
| ais12.cnf | 1 | 372ms |

**3.二进制数独游戏**

棋盘大小4x4：

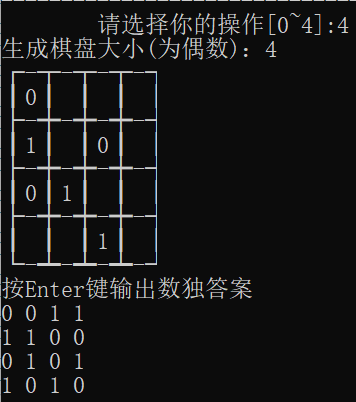
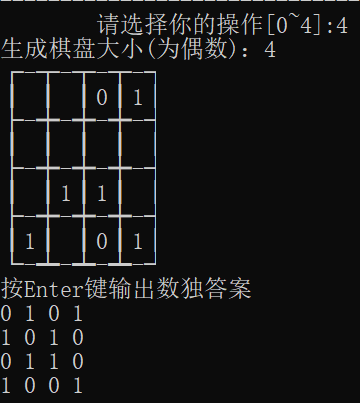
 

图5.3-3 4x4棋局1 图5.3-4 4x4棋局2

棋盘大小6x6：

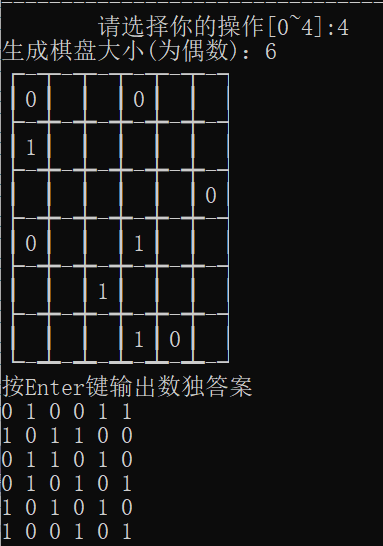
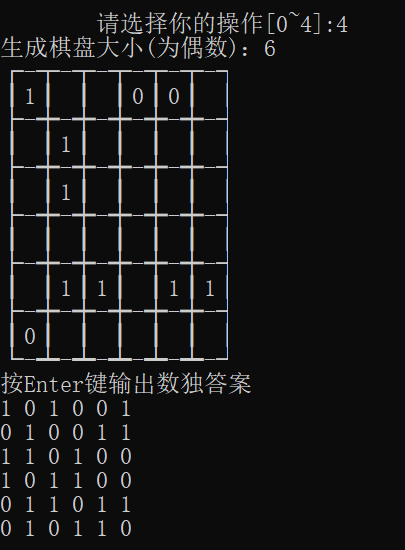
 

图5.3-5 6x6棋局1 图5.3-6 6x6棋局2

棋盘大小8x8：

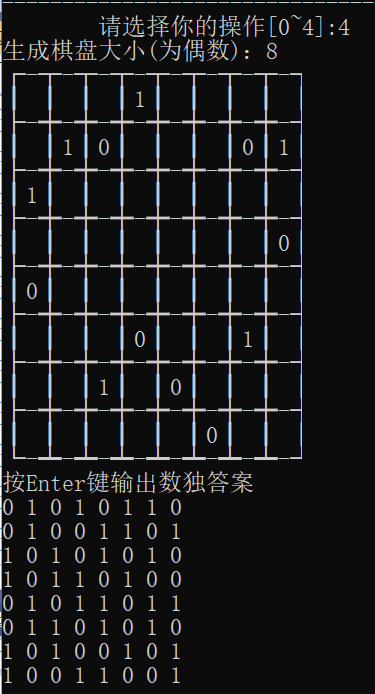
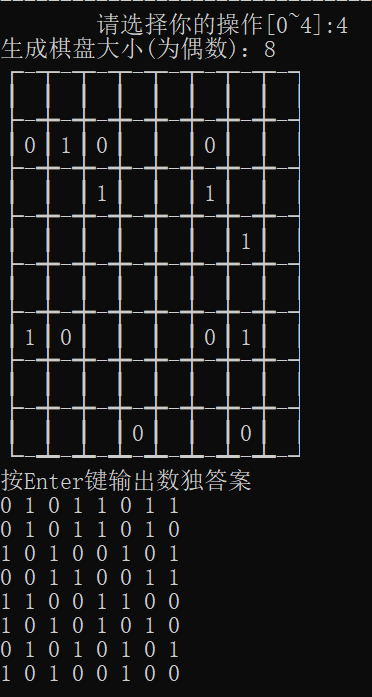
 

图5.3-7 8x8棋局1 图5.3-8 8x8棋局2

## 6 实验小结

课程设计重难点在于DPLL算法的实现，我最开始写的程序能够算出较小的cnf文件，但遇到大文件容易出现内存爆满的情况，后来检查时发现是在DPLL函数递归调用过程中有些申请的空间没有释放掉，导致占用内存十分庞大。后来经过改进，虽然还是会出现内存崩掉的情况，但大部分文件还是可以处理。变元选取策略我最开始选择的是直接选择出现次数最多的文字，后来根据一些资料查阅，改进为启发式策略，但通过实践发现效率仍然不高，最后在摸索和试错当中改进为当前模式，先找出现次数最多的正文字，然后再找负文字，这种策略下大部分文件都能快速得到解，但对一些特殊的文件和较大的文件，解起来十分慢也基本上解不出来，因为会出现内存不够的情况。在二进制数独游戏中，采用的是最简单最直接的随机生成和随机挖洞法，但是没有实现玩家自由填入数字操作，而是直接生成一个棋局和最后答案，让玩家先自己在纸上写好答案再和结果对比。

# 附录 程序源代码

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

int boolCount; //布尔变元数量

int clauseCount; //子句数量

char fileName[100]; //文件名

//十字链表结构体

typedef struct SATNode{

int data; //数据域

SATNode\* next;

}SATNode;

typedef struct SATList {

SATNode\* head; //表头

SATList\* next;

}SATList;

//函数声明

int ReadFile(SATList\*& cnf);

void destroyClause(SATList\*& cnf);

int isUnitClause(SATNode\* cnf);

int evaluateClause(SATNode\* cnf, int v[]);

int removeClause(SATList\*& cnf, SATList\*& root);

int removeNode(SATNode\*& cnf, SATNode\*& head);

int addClause(SATList\* cnf, SATList\*& root);

int emptyClause(SATList\* cnf);

int DPLL(SATList\*& cnf, int value[]);

void CopyClause(SATList\*& a, SATList\* b);

int WriteFile(int result, double time, int value[]);

void CreateBinary(int size);

int SolvePuzzle(int chess[], int size);

//函数定义

/\*

\* 函数名称: main

\* 接受参数: void

\* 函数功能: 主函数

\* 返回值: int

\*/

int main(void)

{

SATList\* CNFList = NULL, \* lp;

SATNode\* tp;

clock\_t start, finish; //设置时间变量

double time;

int op = 1, i, result;

int\* value; //保存结果

while (op)

{

system("cls"); printf("\n\n");

printf(" 功能菜单 \n");

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

printf("1.读取cnf文件 2.遍历输出每个子句\n");

printf("3.DPLL求解算例并保存 4.二进制数独游戏\n");

printf("0.退出\n");

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

printf(" 请选择你的操作[0~4]:");

scanf\_s("%d", &op);

switch (op)

{

case 1:

printf("输入要读取的cnf文件:");

scanf\_s("%s", fileName, 100);

ReadFile(CNFList);

getchar(); getchar();

break;

case 2:

if (CNFList == NULL) printf("未导入文件\n");

else

{

printf("cnf子句如下：\n");

for (lp = CNFList; lp != NULL; lp = lp->next)

{

for (tp = lp->head; tp != NULL; tp = tp->next)

{

printf("%d ", tp->data);

}

printf("\n");

}

}

getchar(); getchar();

break;

case 3:

if (CNFList == NULL) printf("未导入文件\n");

else

{

value = (int\*)malloc(sizeof(int) \* (boolCount + 1));

for (i = 1; i <= boolCount; i++) value[i] = 1; //初始化，均赋为1

start = clock(); //计时开始;

result = DPLL(CNFList, value);

finish = clock(); //结束

printf("求解结果：%d\n", result);

if (result == 1)

{

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

{

if (value[i] == 1) printf("%d ", i);

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

}

printf("\n");

}

time = (double)(finish - start) / CLOCKS\_PER\_SEC;//计算运行时间

printf("运行时间=%lfms\n", time \* 1000);//输出运行时间

if (WriteFile(result, time, value) == 1)

printf("结果已保存至同名文件.res\n");

else printf("结果保存失败\n");

}

getchar(); getchar();

break;

case 4:

printf("生成棋盘大小(为偶数)：");

scanf\_s("%d", &i);

CreateBinary(i);

getchar();

break;

case 0:

break;

}

}

return 0;

}

/\*

\* 函数名称: ReadFile

\* 接受参数: SATList\*&

\* 函数功能: 用文件指针fp打开用户指定的文件，并读取文件内容保存到给定参数中，读取成功返回1，失败返回0

\* 返回值: int

\*/

int ReadFile(SATList\*& cnf)

{

FILE\* fp;

char ch;

int number, i;

SATList\* lp;

SATNode\* tp;

if (fopen\_s(&fp, fileName, "r")) {

printf("文件打开失败!\n");

return 0;

}

while ((ch = getc(fp)) == 'c') {

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

continue; //弃去一整行

} //运行到此，已经读取了字符p

getc(fp); getc(fp); getc(fp); getc(fp); //弃去cnf三个字母

fscanf\_s(fp, "%d", &boolCount); //p后的第1个数值是布尔变元数量

fscanf\_s(fp, "%d", &clauseCount); //p后的第2个数值是子句数量

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

cnf->next = NULL;

cnf->head = (SATNode\*)malloc(sizeof(SATNode));

cnf->head->next = NULL;

lp = cnf;

tp = cnf->head;

//创建数量为clauseCount的子句, i为计数器

for (i = 0; i < clauseCount; i++, lp = lp->next, tp = lp->head)

{

fscanf\_s(fp, "%d", &number);

for (; number != 0; tp = tp->next)

{

tp->data = number; //数据域赋值

tp->next = (SATNode\*)malloc(sizeof(SATNode)); //开辟新结点

fscanf\_s(fp, "%d", &number);

if (number == 0) tp->next = NULL;

}

lp->next = (SATList\*)malloc(sizeof(SATList)); //开辟新表

lp->next->head = (SATNode\*)malloc(sizeof(SATNode));

if (i == clauseCount - 1)

{

lp->next = NULL;

break;

}

}

printf("读取完毕\n");

fclose(fp);

return 1;

}

/\*

\* 函数名称: destroyClause

\* 接受参数: SATList\*&

\* 函数功能: 销毁链表

\* 返回值: int

\*/

void destroyClause(SATList\*& cnf)

{

SATList\* lp1, \* lp2;

SATNode\* tp1, \* tp2;

for (lp1 = cnf; lp1 != NULL; lp1 = lp2)

{

lp2 = lp1->next;

for (tp1 = lp1->head; tp1 != NULL; tp1 = tp2)

{

tp2 = tp1->next;

free(tp1);

}

free(lp1);

}

cnf = NULL;

}

/\*

\* 函数名称: isUnitClause

\* 接受参数: SATNode\*

\* 函数功能: 判断是否为单子句，是返回1，不是返回0

\* 返回值: int

\*/

int isUnitClause(SATNode\* cnf)

{

if (cnf != NULL && cnf->next == NULL)

return 1;

else

return 0;

}

/\*

\* 函数名称: evaluateClause

\* 接受参数: SATList\*

\* 函数功能: 评估子句的真假状态，真返回1，假返回0

\* 返回值: int

\*/

int evaluateClause(SATNode\* cnf,int v[])

{

SATNode\* tp = cnf;

while (tp != NULL)

{

if (tp->data > 0 && v[tp->data] == 1 ||

tp->data < 0 && v[-tp->data] == 0)

return 1;

}

return 0;

}

/\*

\* 函数名称: removeClause

\* 接受参数: SATList\*,SATList\*

\* 函数功能: 在已有的十字链表中删除指定的子句，删除成功返回1，失败返回0

\* 返回值: int

\*/

int removeClause(SATList\*& cnf, SATList\*& root)

{

SATList\* lp = root;

if (lp == cnf) root = root->next; //删除为头

else

{

while (lp != NULL && lp->next != cnf) lp = lp->next;

lp->next = lp->next->next;

}

free(cnf);

cnf = NULL;

return 1;

}

/\*

\* 函数名称: removeNote

\* 接受参数: SATNode\*,SATNode\*

\* 函数功能: 在指定的子句中删除指定的文字，删除成功返回1，失败返回0

\* 返回值: int

\*/

int removeNode(SATNode\*& cnf, SATNode\*& head)

{

SATNode\* lp = head;

if (lp == cnf) head = head->next; //删除为头

else

{

while (lp != NULL && lp->next != cnf) lp = lp->next;

lp->next = lp->next->next;

}

free(cnf);

cnf = NULL;

return 1;

}

/\*

\* 函数名称: addClause

\* 接受参数: SATList\*,SATList\*

\* 函数功能: 在已有的十字链表中添加指定的子句，添加成功返回1，失败返回0

\* 返回值: int

\*/

int addClause(SATList\* cnf, SATList\*& root)

{

//直接插入在表头

if (cnf != NULL)

{

cnf->next = root;

root = cnf;

return 1;

}

return 0;

}

/\*

\* 函数名称: emptyClause

\* 接受参数: SATList\*

\* 函数功能: 判断是否含有空子句，是返回1，不是返回0

\* 返回值: int

\*/

int emptyClause(SATList\* cnf)

{

SATList\* lp = cnf;

while (lp != NULL)

{

if (lp->head == NULL) return 1;

lp = lp->next;

}

return 0;

}

/\*

\* 函数名称: CopyClause

\* 接受参数: SATList\*,SATList\*

\* 函数功能: 将链表b的值复制到链表a中

\* 返回值: void

\*/

void CopyClause(SATList\*& a, SATList\* b)

{

SATList\* lpa,\*lpb;

SATNode\* tpa,\*tpb;

a = (SATList\*)malloc(sizeof(SATList));

a->head = (SATNode\*)malloc(sizeof(SATNode));

a->next = NULL;

a->head->next = NULL;

for (lpb = b, lpa = a; lpb != NULL; lpb = lpb->next, lpa = lpa->next)

{

for (tpb = lpb->head, tpa = lpa->head; tpb != NULL; tpb = tpb->next, tpa = tpa->next)

{

tpa->data = tpb->data;

tpa->next = (SATNode\*)malloc(sizeof(SATNode));

tpa->next->next = NULL;

if (tpb->next == NULL)

{

free(tpa->next);

tpa->next = NULL;

}

}

lpa->next = (SATList\*)malloc(sizeof(SATList));

lpa->next->head = (SATNode\*)malloc(sizeof(SATNode));

lpa->next->next = NULL;

lpa->next->head->next = NULL;

if (lpb->next == NULL)

{

free(lpa->next->head);

free(lpa->next);

lpa->next = NULL;

}

}

}

/\*

\* 函数名称: DPLL

\* 接受参数: SATList \*

\* 函数功能: 求解SAT问题，给出满足条件时的一个式子,若有解则返回1，无解返回0

\* 返回值: int

\*/

int DPLL(SATList\*& cnf, int value[])

{

SATList\* tp = cnf, \* lp = cnf, \* sp;

SATNode\* dp;

int\* count, i, MaxWord, max, re; //count记录每个文字出现次数,MaxWord记录出现最多次数的文字

count = (int\*)malloc(sizeof(int) \* (boolCount \* 2 + 1));

FIND: while (tp != NULL && isUnitClause(tp->head) == 0) tp = tp->next; //找到表中的单子句

if (tp != NULL)

{

//单子句规则简化

if (tp->head->data > 0) value[tp->head->data] = 1;

else value[-tp->head->data] = 0;

re = tp->head->data;

for (lp = cnf; lp != NULL; lp = sp)

{

sp = lp->next;

//查找含有核心文字的句子

for (dp = lp->head; dp != NULL; dp = dp->next)

{

if (dp->data == re)

{

removeClause(lp, cnf); //删除子句，简化式子

break;

}

if (dp->data == -re)

{

removeNode(dp, lp->head); //删除文字，简化子句

break;

}

}

}

//极简化规则简化后

if (cnf == NULL)

{

free(count);

return 1;

}

else if (emptyClause(cnf))

{

free(count);

destroyClause(cnf);

return 0;

}

tp = cnf;

goto FIND; //继续简化

}

for (i = 0; i <= boolCount \* 2; i++) count[i] = 0; //初始化

//计算子句中各文字出现次数

for (lp = cnf; lp != NULL; lp = lp->next)

{

for (dp = lp->head; dp != NULL; dp = dp->next)

{

if (dp->data > 0) count[dp->data]++;

else count[boolCount - dp->data]++;

}

}

max = 0;

//找到出现次数最多的正文字

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

{

if (max < count[i])

{

max = count[i];

MaxWord = i;

}

}

if (max == 0)

{

//若没有出现正文字,找到出现次数最多的负文字

for (i = boolCount + 1; i <= boolCount \* 2; i++)

{

if (max < count[i])

{

max = count[i];

MaxWord = -i;

}

}

}

free(count);

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->data = MaxWord;

lp->head->next = NULL;

lp->next = NULL;

CopyClause(tp, cnf);

addClause(lp, tp);

if (DPLL(tp, value) == 1) return 1; //在第一分支中搜索

destroyClause(tp);

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->data = -MaxWord;

lp->head->next = NULL;

lp->next = NULL;

addClause(lp, cnf);

re = DPLL(cnf, value); //回溯到执行分支策略的初态进入另一分支

destroyClause(cnf);

return re;

}

/\*

\* 函数名称: WriteFile

\* 接受参数: int,int,int[]

\* 函数功能: 将运行结果保存至同名文件，文件拓展名为.res,保存成功返回1，失败返回0

\* 返回值: int

\*/

int WriteFile(int result, double time, int value[])

{

FILE\* fp;

int i;

for (i = 0; fileName[i] != '\0'; i++)

{

//修改拓展名

if (fileName[i] == '.' && fileName[i + 4] == '\0')

{

fileName[i + 1] = 'r';

fileName[i + 2] = 'e';

fileName[i + 3] = 's';

break;

}

}

if (fopen\_s(&fp, fileName, "w")) {

printf("文件打开失败!\n");

return 0;

}

fprintf(fp, "s %d\nv ", result); //求解结果

if (result == 1)

{

//保存解值

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

{

if (value[i] == 1) fprintf(fp, "%d ", i);

else fprintf(fp, "%d ", -i);

}

}

fprintf(fp, "\nt %lf", time \* 1000); //运行时间/毫秒

fclose(fp);

return 1;

}

/\*

\* 函数名称: CreateBinary

\* 接受参数: int

\* 函数功能: 按照指定大小生成棋盘

\* 返回值: void

\*/

void CreateBinary(int size)

{

int\* chess, i, j = 0;

//开辟size\*size大小的数组

chess = (int\*)malloc(sizeof(int) \* (size \* size + 1));

for (i = 0; i <= size\*size; i++) chess[i] = -1; //初始化

srand(time(0)); //时间随机

chess[rand() % (size \* size + 1)] = rand() % 2; //随机给定棋盘一初始条件

SolvePuzzle(chess, size); //生成二进制数独棋盘

for (i = 0; i <= rand() % (size / 2) + size \* (size - 2); i++)

{

//随机选择size \* (size - 2)到size \* (size - 2)+(size / 2)个数字清空

while (j == 0 || chess[j] == -1) j = rand() % (size \* size + 1); //选择合适的数字

chess[j] = -1;

}

for (j = 0, i = 0; j <= size \* size || i <= (size+1) \* (size+1);)

{

if (j % size == 0)

{

//打印棋盘

if (i % (size + 1) == 0 && i > 0) printf("┃\n");

i++;

if (i == 1) printf("┌-");

else if (i < size + 1) printf("-┬-");

else if (i == size + 1) printf("-┐");

else if (i == (size + 1) \* size + 1) printf("└-");

else if (i == (size + 1) \* (size + 1)) printf("-┘");

else if (i > (size + 1)\* size) printf("-┷-");

else if (i % (size + 1) == 1) printf("├-");

else if (i % (size + 1) == 0) printf("-┤");

else printf("-┿-");

}

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

{

//输出数独

if (j % size == 0) printf("\n");

j++;

if (j > size\* size) break;

if (chess[j] == -1) printf("┃ ");

else printf("┃ %d", chess[j]);

}

}

printf("按Enter键输出数独答案");

getchar(); getchar();

SolvePuzzle(chess, size); //求解答案

for (i = 1; i <= size \* size; i++)

{

printf("%d ", chess[i]);

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

}

free(chess);

}

/\*

\* 函数名称: SolvePuzzle

\* 接受参数: int[],int

\* 函数功能: 求解输入棋盘，有解返回1，无解返回0

\* 返回值: int

\*/

int SolvePuzzle(int chess[], int size)

{

SATList\* cnf = NULL, \* lp;

SATNode\* dp;

int\* remember, i, j, k, rol;

boolCount = size \* size;

//添加单子句

for (i = 1; i <= size \* size; i++)

{

if (chess[i] == 0)

{

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

lp->head->data = -i;

addClause(lp, cnf);

}

else if (chess[i] == 1)

{

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

lp->head->data = i;

addClause(lp, cnf);

}

}

//约束1：不出现连续三个重复数字

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

{

for (j = 0; j < size - 2; j++)

{

//横向，正文字

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = i \* size + j + 1;

lp->head->next->data = i \* size + j + 2;

lp->head->next->next->data = i \* size + j + 3;

addClause(lp, cnf);

//横向，负文字

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i \* size + j + 1);

lp->head->next->data = -(i \* size + j + 2);

lp->head->next->next->data = -(i \* size + j + 3);

addClause(lp, cnf);

//纵向，正文字

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = i + j \* size + 1;

lp->head->next->data = i + (j + 1) \* size + 1;

lp->head->next->next->data = i + (j + 2) \* size + 1;

addClause(lp, cnf);

//纵向，负文字

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i + j \* size + 1);

lp->head->next->data = -(i + (j + 1) \* size + 1);

lp->head->next->next->data = -(i + (j + 2) \* size + 1);

addClause(lp, cnf);

}

}

//约束2：在每一行、每一列中1与0的个数相同

remember = (int\*)malloc(sizeof(int) \* (size / 2 + 1));

//每一行

for (rol = 0; rol < size; rol++)

{

for (i = 0; i < size / 2 + 1; i++) remember[i] = i + 1; //初始化

COMBINATION1:for (i = size / 2; remember[i] <= size; remember[i]++)

{

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (j = 0, dp = lp->head; j < size / 2 + 1; j++, dp = dp->next)

{

dp->data = remember[j] + rol \* size;

if (j == size / 2) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (j = 0, dp = lp->head; j < size / 2 + 1; j++, dp = dp->next)

{

dp->data = -(remember[j] + rol \* size);

if (j == size / 2) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

}

for (i = size / 2; i >= 0 && remember[i] >= size / 2 + i; i--); //找到达到饱和的最高位

if (i <= 0) continue; //该行组合序列全部排完,进入下一行

remember[i]++;

for (j = i + 1; j < size / 2 + 1; j++) remember[j] = remember[j - 1] + 1; //序列后移

goto COMBINATION1;

}

//每一列

for (rol = 1; rol <= size; rol++)

{

for (i = 0; i < size / 2 + 1; i++) remember[i] = i; //初始化

COMBINATION2:for (i = size / 2; remember[i] < size; remember[i]++)

{

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (j = 0, dp = lp->head; j < size / 2 + 1; j++, dp = dp->next)

{

dp->data = remember[j] \* size + rol;

if (j == size / 2) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (j = 0, dp = lp->head; j < size / 2 + 1; j++, dp = dp->next)

{

dp->data = -(remember[j] \* size + rol);

if (j == size / 2) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

}

for (i = size / 2; i >= 0 && remember[i] > size / 2 + i; i--); //找到达到饱和的最高位

if (i <= 0) continue; //该列组合序列全部排完,进入下一列

remember[i]++;

for (j = i + 1; j < size / 2 + 1; j++) remember[j] = remember[j - 1] + 1; //序列后移

goto COMBINATION2;

}

//约束3：不存在重复的行与重复的列

//不重复行：

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

{

for (j = i + 1; j < size; j++)

{

rol = boolCount; //记录添加变元之前变元数量

for (k = 1; k <= size; k++)

{

//引入前一个新变元：

boolCount++;

//语句1：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = i \* size + k;

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句2：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = -(j \* size + k);

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句3：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i \* size + k);

lp->head->next->data = j \* size + k;

lp->head->next->next->data = boolCount;

addClause(lp, cnf);

//引入后一个新变元：

boolCount++;

//语句1：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i \* size + k);

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句2：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = j \* size + k;

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句3：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = i \* size + k;

lp->head->next->data = -(j \* size + k);

lp->head->next->next->data = boolCount;

addClause(lp, cnf);

}

//添加长句：不重复行满足的关系

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (k = rol + 1, dp = lp->head; k <= boolCount; k++, dp = dp->next)

{

dp->data = k;

if (k == boolCount) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

}

}

//不重复列

for (i = 1; i <= size - 1; i++)

{

for (j = i + 1; j <= size; j++)

{

rol = boolCount; //记录添加变元之前变元数量

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

{

//引入前一个新变元：

boolCount++;

//语句1：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = i + k \* size;

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句2：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = -(j + k \* size);

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句3：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i + k \* size);

lp->head->next->data = j + k \* size;

lp->head->next->next->data = boolCount;

addClause(lp, cnf);

//引入后一个新变元：

boolCount++;

//语句1：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = -(i + k \* size);

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句2：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = NULL;

lp->next = NULL;

lp->head->data = j + k \* size;

lp->head->next->data = -boolCount;

addClause(lp, cnf);

//语句3：

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next->next->next = NULL;

lp->next = NULL;

lp->head->data = i + k \* size;

lp->head->next->data = -(j + k \* size);

lp->head->next->next->data = boolCount;

addClause(lp, cnf);

}

//添加长句：不重复行满足的关系

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

lp->head = (SATNode\*)malloc(sizeof(SATNode));

lp->head->next = NULL;

lp->next = NULL;

for (k = rol + 1, dp = lp->head; k <= boolCount; k++, dp = dp->next)

{

dp->data = k;

if (k == boolCount) break;

dp->next = (SATNode\*)malloc(sizeof(SATNode));

dp->next->next = NULL;

}

addClause(lp, cnf);

}

}

free(remember);

remember = (int\*)malloc(sizeof(int) \* (boolCount + 1));

for (i = 1; i <= boolCount; i++) remember[i] = 1; //初始化

if (DPLL(cnf, remember) == 1)

{

for (i = 1; i <= size \* size; i++) chess[i] = remember[i];

free(remember);

destroyClause(cnf);

return 1;

}

else

{

free(remember);

destroyClause(cnf);

return 0;

}

}